El lenguaje de Programación C

Este es un ejemplar amorosamente wikificado por ~peron de *El Lenguaje de Programación C*, 2da Edición, de B. Kerningham y D. Ritchie.

Prefacio

El mundo de la computación ha sufrido una revolución desde la publicación, en 1978, de *El lenguaje de programación C*. Las grandes computadoras son ahora mucho más grandes, y las computadoras personales tienen capacidades que rivalizan con los mainframes de hace una década. También el lenguaje C ha cambiado en ese tiempo, aunque sólo en forma modesta, y se ha extendido más allá de lo que fueron sus orígenes como el lenguaje del sistema operativo UNIX.

La creciente popularidad de C , los cambios en el lenguaje a lo largo de los años, y la creación de compiladores por grupos no involucrados en su diseño, se combinaron para demostrar la necesidad de una definición del lenguaje más precisa y contemporánea que la que proporcionó la primera edición de este libro.

En 1983, el American National Standards Institute (ANSI) estableció un comité cuyos propósitos eran producir “una definición no ambigua del lenguaje C, independiente de la máquina” , cuidando la conservación de su espíritu. El resultado es el estándar ANSI para el lenguaje C.

El estándar formaliza construcciones sugeridas pero no descritas en la primera edición, particularmente la asignación de estructura y las enumeraciones. Proporciona una nueva forma de declaración de funciones, que permite revisar comparativamente su definición y uso. Especifica una biblioteca estándar, con un conjunto extensivo de funciones para realizar la entrada y salida, la administración de memoria, la manipulación de cadenas y tareas semejantes. Precisa el comportamiento de características que no se mencionaron en la definición original, y al mismo tiempo establece explícitamente cuáles aspectos del lenguaje tienen aún dependencia de máquina.

Esta segunda edición de **El lenguaje de programación C** lo describe tal como lo definió el estándar ANSI. (En el momento de escribir está edición, el estándar se encontraba en la etapa final de revisión; se esperaba su aprobación a finales de 1988. Las diferencias entre lo que se ha descrito aquí y la forma final deberán ser mínimas.) Aunque hemos hecho anotaciones en los lugares donde el lenguaje ha evolucionado, preferimos escribir exclusivamente en la nueva forma. En general esto no hace una diferencia significativa; el cambio más visible es la nueva forma de declaración y definición de funciones. Los modernos compiladores manejan ya la mayoría de las posibilidades del estándar.

Hemos tratado de mantener la brevedad de la primera edición. El lenguaje C no es grande, y no le está bien un gran libro. Hemos mejorado la exposición de características críticas, como los apuntadores, que son parte central en la programación con C. Hemos redefinido los ejemplos originales y agregamos ejemplos nuevos en varios capítulos. Por ejemplo, se aumentó el tratamiento de declaraciones complicadas con programas que convierten declaraciones en palabras y viceversa. Como antes, todos los ejemplos se han probado directamente a partir del texto, el cual está diseñado de manera que lo pueda leer la máquina.

El apéndice A, manual de referencia, no es el estándar, sino que nuestra intención fue trasladar la esencia del estándar a un espacio más pequeño. Está hecho con el ánimo de que proporcione una fácil com prensión para los programadores, pero no como una definición del lenguaje para quienes escriben compiladores —ese papel propiamente le corresponde al estándar en sí. El apéndice B es un resumen de las posibilidades de la biblioteca estándar. También tiene el propósito de ser una referencia para programadores, no para implantadores. En el apéndice C se ofrece un resumen de los cambios de la versión original.

Como mencionamos en el prefacio a la primera edición, C “se lleva bien, en la medida en que aumenta nuestra experiencia con él” . Con una década más de experiencia, aún lo sentimos así. Deseamos que este libro le ayude a aprender el lenguaje C y también cómo usarlo.

Tenemos un profundo reconocimiento hacia los amigos que nos ayudaron a producir esta segunda edición. Jo n Bentley, Doug Gwyn, Doug Mcllroy, Peter Nelson y Rob Pike nos dieron valiosos comentarios sobre casi cada página del borrador de este manuscrito. Estamos agradecidos por la cuidadosa lectura de AI Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Alien Holub, Andrew H um e, Dave Kristol, John Linderm an, Dave Prosser, Gene Spafford, y Chris Van Wyk. También recibimos útiles sugerencias de Bill Cheswick, Mark Kernighan, Andy Koening, Robin Lake, Tom London, Jim Reeds, Clovis Tondo y Peter Weinberger. Dave Prosser respondió muchas preguntas detalladas acerca del estándar ANSI. Utilizamos extensivamente el intérprete de C++ de Bjarne Stroustrup, para la prueba local de nuestros programas, y Dave Kristol nos ofreció un compilador ANSI C para las pruebas finales. Rich Drechsler nos ayudó grandemente con la composición.

Nuestro sincero agradecimiento a todos.

Brian W. Kernighan Dennis M. Ritchie

Introducción

C es un lenguaje de programación de propósito general que ha sido estrechamente asociado con el sistema UNIX en donde fue desarrollado puesto que tanto el sistema como los programas que corren en él están escritos en lenguaje C. Sin embargo, este lenguaje no está ligado a ningún sistema operativo ni a ninguna máquina, y aunque se le llama “lenguaje de programación de sistemas” debido a su utilidad para escribir compiladores y sistemas operativos, se utiliza con igual eficacia para escribir importantes programas en diversas disciplinas.

Muchas de las ideas importantes de C provienen del lenguaje BCPL, desarrollado por Martin Richards. La influencia de BCPL sobre C se continuó indirectamente a través del lenguaje B, el cual fue escrito por Ken Thompson en 1970 para el primer sistema UNIX de la DEC PDP-7.

BCPL y B son lenguajes “carentes de tipos”. En contraste, C proporciona una variedad de tipos de datos. Los tipos fundamentales son caracteres, enteros y números de punto flotante de varios tamaños. Además, existe una jerarquía de tipos de datos derivados, creados con apuntadores, arreglos, estructuras y uniones. Las expresiones se forman a partir de operadores y operandos; cualquier expresión, incluyendo una asignación o una llamada a función, puede ser una proposición. Los apuntadores proporcionan una aritmética de direcciones independiente de la máquina.

C proporciona las construcciones fundamentales de control de flujo que se requieren en programas bien estructurados: agrupación de proposiciones, toma de decisiones (if-else), selección de un caso entre un conjunto de ellos (switch), iteración con la condición de paro en la parte superior (while, for) o en la parte inferior (do), y terminación prematura de ciclos (break).

Las funciones pueden regresar valores de tipos básicos, estructuras, uniones o apuntadores. Cualquier función puede ser llamada recursivamente. Las variables locales son normalmente “automáticas”, o creadas de nuevo con cada invocación. La definición de una función no puede estar anidada, pero las variables pueden estar declaradas en una modalidad estructurada por bloques. Las funciones de un programa en C pueden existir en archivos fuente separados, que se compilan de manera separada. Las variables pueden ser internas a una función, externas pero conocidas sólo dentro de un archivo fuente, o visibles al programa completo.

Un paso de preprocesamiento realiza substitución de macros en el texto del programa, inclusión de otros archivos fuente y compilación condicional.

C es un lenguaje de relativo “bajo nivel”. Esta caracterización no es peyorativa, simplemente significa que C trata con el mismo tipo de objetos que la mayoría de las computadoras, llámense caracteres, números y

direcciones. Estos pueden ser combinados y cambiados de sitio con los operadores aritméticos y lógicos implantados por máquinas reales.

C no proporciona operaciones para tratar directamente con objetos compuestos, tales como cadenas de caracteres, conjuntos, listas o arreglos. No existen operaciones que manipulen un arreglo o una cadena completa, aunque las estructuras pueden copiarse como una unidad. El lenguaje no define ninguna facilidad para asignación de almacenamiento que no sea la de definición estática y la disciplina de pilas provista por las variables locales de funciones; no emplea *heap* ni recolector de basura. Finalmente, C en sí mismo no proporciona capacidades de entrada/salida; no hay proposiciones READ o WRITE, ni métodos propios de acceso a archivos. Todos esos mecanismos de alto nivel deben ser proporcionados por funciones llamadas explícitamente.

De manera semejante, C solamente ofrece un control de flujo franco, y lineal: condiciones, ciclos, agrupamientos y subprogramas, pero no multiprogramación, operaciones paralelas, sincronización ni co-rutinas.

Aunque la ausencia de alguna de esas capacidades puede parecer como una grave deficiencia (“¿significa que se tiene que llamar a una función para comparar dos cadenas de caracteres?”), el mantener al lenguaje de un tamaño modesto tiene beneficios reales. Puesto que C es relativamente pequeño, se puede describir en un pequeño espacio y aprenderse con rapidez. Un programador puede razonablemente esperar conocer, entender y utilizar en verdad la totalidad del lenguaje.

Por muchos años, la definición de C fue el manual de referencia de la primera edición de El lenguaje de programación C. En 1983, el American National Standards Institute (ANSI) estableció un comité para proporcionar una moderna y comprensible definición de C. La definición resultante, el estándar ANSI o “ANSI C” , se esperaba fuera aprobada a fines de 1988. La mayoría de las características del estándar ya se encuentran soportadas por compiladores modernos.

El estándar está basado en el manual de referencia original. El lenguaje ha cambiado relativamente poco; uno de los propósitos del estándar fue asegurar que la mayoría de los programas existentes pudiesen permanecer válidos o, al menos, que los compiladores pudieran producir mensajes de advertencia acerca del nuevo comportamiento.

Para la mayoría de los programadores, el cambio más importante es una nueva sintaxis para declarar y definir funciones. Una declaración de función ahora puede incluir una descripción de los argumentos de la función; la sintaxis de la definición cambia para coincidir. Esta información extra permite que los compiladores detecten más fácilmente los errores causados por argumentos que no coinciden; de acuerdo con nuestra experiencia, es una adición muy útil al lenguaje.

Existen otros cambios de menor escala en el lenguaje. La asignación de estructuras y enumeraciones, que ha estado ampliamente disponible, es ahora parte oficial del lenguaje. Los cálculos de punto flotante pueden ahora realizarse con precisión sencilla. Las propiedades de la aritmética, especialmente para tipos sin signo, están esclarecidas. El preprocesador es más elaborado. La mayor parte de esos cambios sólo tendrán efectos secundarios para la mayoría de los programadores.

Una segunda contribución significativa del estándar es la definición de una biblioteca que acompañe a C. Esta especifica funciones para tener acceso al sistema operativo (por ejemplo, leer de archivos y escribir en ellos), entrada y salida con formato, asignación de memoria, manipulación de cadenas y otras actividades semejantes. Una colección de encabezadores (headers) estándar proporcionan un acceso uniforme a las declaraciones de funciones y tipos de datos. Los programas que utilizan esta biblioteca para interactuar con un sistema anfitrión están asegurados de un comportamiento compatible. La mayor parte de la biblioteca está estrechamente modelada con base en la “biblioteca E/S estándar“ del sistema UNIX. Esta biblioteca se describió en la primera edición y ha sido también ampliamente utilizada en otros sistemas. De nuevo, la mayoría de los programadores no notarán mucho el cambio.

Debido a que los tipos de datos y estructuras de control provistas por C son manejadas directamente por la mayoría de las computadoras, la biblioteca de ejecución (run-time) requerida para implantar programas autocontenidos es pequeña. Las funciones de la biblioteca estándar únicamente se llaman en forma explícita, de manera que se pueden evitar cuando no se necesitan. La mayor parte puede escribirse en C, y excepto por detalles ocultos del sistema operativo, ellas mismas son portátiles.

Aunque C coincide con las capacidades de muchas computadoras, es independiente de cualquier arquitectura. Con un poco de cuidado es fácil escribir programas portátiles, esto es, programas que puedan correr sin cambios en una variedad de máquinas. El estándar explica los problemas de la transportabilidad, y prescribe un conjunto de constantes que caracterizan a la máquina en la que se ejecuta el programa.

C no es un lenguaje fuertemente tipificado, sino que, al evolucionar, su verificación de tipos ha sido reforzada. La definición original de C desaprobó - pero permitió - el intercambio de apuntadores y enteros; esto se ha eliminado y el estándar ahora requiere la adecuada declaración y la conversión explícita que ya ha sido obligada por los buenos compiladores. La nueva declaración de funciones es otro paso en esta dirección. Los compiladores advertirán de la mayoría de los errores de tipo, y no hay conversión automática de tipos de datos incompatibles.

Sin embargo, C mantiene la filosofía básica de que los programadores saben lo Que están haciendo; sólo requiere que establezcan sus intenciones en forma explícita.

Como cualquier otro lenguaje, C tiene sus defectos. Algunos de los operadores tienen la precedencia equivocada; algunos elementos de la sintaxis pueden ser mejores. A pesar de todo, C ha probado ser un lenguaje extremadamente efectivo y expresivo p ara una am plia variedad de program as de aplicación.

El libro está organizado como sigue. El capítulo 1 es una introducción orientada a la parte central de C. El propósito es hacer que el lector se inicie tan pronto como le sea posible, puesto que creemos firmemente que la forma de aprender un nuevo lenguaje es escribir programas en él. La introducción supone un conocimiento práctico de los elementos básicos de la programación; no hay una explicación de computadoras, de compilación, ni del significado de una expresión como n = n + 1. Aunque hemos tratado de mostrar técnicas útiles de programación en donde fue posible, la intención del libro no es la de ser un texto de consulta sobre estructuras de datos y algoritmos; cuando nos vimos forzados a hacer una elección, nos hemos concentrado en el lenguaje.

En los capítulos del 2 al 6 se discuten varios aspectos de C en mayor detalle y más formalmente de lo que se hace en el capítulo 1, aunque el énfasis está aún en los ejemplos de programas completos, más que en fragmentos aislados. El capítulo 2 trata de los tipos básicos de datos, operaciones y expresiones. El capítulo 3 trata sobre control de flujo: if-else, switch, while, for, etc. En el capítulo 4 se cubren funciones y la estructura de un programa — variables externas, reglas de alcance, archivos fuente múltiples y otros aspectos— y también abarca al preprocesador. El capítulo 5 discute sobre apuntadores y aritmética de direcciones.

El capítulo 6 cubre estructuras y uniones.

El capítulo 7 describe la biblioteca estándar, la cual proporciona una interfaz común con el sistema operativo. Esta biblioteca está definida por el estándar ANSI y se intenta que se tenga en todas las máquinas que manejan C; así, los programas que la usen para entrada, salida y otros accesos al sistema operativo se puedan transportar de un sistema a otro sin cambios.

El capítulo 8 describe una interfaz entre los programas en C y el sistema operativo UNIX, concentrándose en entrada/salida, el sistema de archivos y la asignación de memoria. Aunque algo de este capítulo es específico de sistemas UNIX, los programadores que usen otros sistemas de todas maneras encontrarán aquí material de utilidad, incluyendo alguna comprensión acerca de cómo está implantada una versión de la biblioteca estándar, así como sugerencias para obtener un código portátil.

El apéndice A contiene un manual de consulta del lenguaje. El informe oficial de la sintaxis y la semántica de C es en sí el estándar ANSI. Ese documento, sin embargo, está principalmente pensado para quienes escriben compiladores. El manual de consulta de este libro transmite la definición del lenguaje en una forma más concisa y sin el mismo estilo legalista. El apéndice

B es un resumen de la biblioteca estándar, de nuevo más para usuarios que para implantadores. El apéndice C es un breve resumen de los cambios del lenguaje original. Aunque, en caso de duda, el estándar y el compilador en uso quedan como las autoridades finales sobre el lenguaje.

Capítulo 1: Introducción General

Comencemos con una introducción rápida a C. Nuestro objetivo es mostrar los elementos esenciales del lenguaje en programas reales, pero sin perdernos en detalles, reglas o excepciones. Por el momento, no intentamos ser completos ni precisos (exceptuando en los ejemplos, que sí lo son). Deseamos llevarlo tan rápido com o sea posible al punto en donde pueda escribir programas útiles, y para hacerlo tenemos que concentrarnos en las bases: variables y constantes, aritmética, control de flujo, funciones y los rudimentos de entrada y salida. Hemos dejado intencionalmente fuera de este capítulo las características de C que son importantes para escribir programas más grandes. Esas características incluyen apuntadores, estructuras, la mayor parte del rico conjunto de operadores de C, varias proposiciones para control de flujo y la biblioteca estándar.

Este enfoque tiene sus inconvenientes. Lo más notorio es que aquí no se encuentra la descripción completa de ninguna característica particular del lenguaje, y la introducción, por su brevedad, puede también resultar confusa. Y debido a que los ejemplos no utilizan la potencia completa de C, no son tan concisos y elegantes como podrían serlo. Hemos tratado de aminorar esos efectos, pero tenga cuidado. Otro inconveniente es que los capítulos posteriores necesariamente repetirán algo de lo expuesto en éste. Esperamos que la repetición, más que molestar, ayude.

En cualquier caso, los programadores con experiencia deben ser capaces de extrapolar del material que se encuentra en este capítulo a sus propias necesidades de programación. Los principiantes deben complementarlo escribiendo pequeños programas semejantes a los aquí expuestos. Ambos grupos pueden utilizar este capítulo como un marco de referencia sobre el cual asociar las descripciones más detalladas que comienzan en el capítulo 2.

1.1 Comencemos

La única forma de aprender un nuevo lenguaje de programación es escribiendo programas con él. El primer programa por escribir es el mismo para todos los lenguajes:

Imprime las palabras

```
Viva Peron!
```

Este es el gran obstáculo; para librarlo debe tener la habilidad de crear el texto del programa de alguna manera, compilarlo con éxito, cargarlo, ejecutarlo y descubrir a dónde fue la salida. Con el dominio de estos detalles mecánicos, todo lo demás es relativamente fácil.

En C, el programa para escribir “`Viva Perón!`” es

```
#include <stdio.h>
```

```
main()
(
    printf("Viva Peron!\n");
}
```

La forma de ejecutar este programa dependerá del sistema que se esté utilizando. Como un ejemplo específico, en el sistema operativo UNIX se debe crear el programa en un archivo cuyo nombre termine con “`.c`” , como `vivaperon.c` , y después compilarlo con la orden

```
cc vivaperon.c
```

Si no se ha cometido algún error, como la omisión de un carácter o escribir algo en forma incorrecta, la compilación se hará sin emitir mensaje alguno, y creará un archivo ejecutable llamado `a.out`. Si se ejecuta `a.out` escribiendo la orden

```
a.out
```

se presentará:

```
Viva Peron!
```

En otros sistemas, las reglas serán diferentes, consúltelo con un experto.

Ahora algunas explicaciones acerca del programa en sí. Un programa en C, cualquiera que sea su tamaño, consta de *funciones* y *variables*. Una función contiene *proposiciones* que especifican las operaciones de cálculo que se van a realizar, y las variables almacenan los valores utilizados durante los cálculos. Las funciones de C son semejantes a las subrutinas y funciones de Fortran o a los procedimientos y funciones de Pascal. Nuestro ejemplo es una función llamada `main`. Normalmente se tiene la libertad de dar cualquier nombre que se desee, pero “`main`” es especial — el programa comienza a ejecutarse al principio de `main`. Esto significa que todo programa debe tener un `main` en algún sitio.

Por lo común `main` llamará a otras funciones que ayuden a realizar su trabajo, algunas que usted ya escribió, y otras de bibliotecas escritas previamente. La primera línea del programa.

```
#include <stdio.h>
```

indica al compilador que debe incluir información acerca de la biblioteca estándar de entrada/salida; esta línea aparece al principio de muchos archivos de código fuente de lenguaje C. La biblioteca estándar está descrita en el capítulo 7 y en el apéndice B.

Un método para comunicar datos entre las funciones es que la función que llama proporcione una lista de valores, llamados *argumentos*, a la función que está invocando. Los paréntesis que están después del nombre de la función encierran a la lista de argumentos. En este ejemplo, `main` está definido para

ser una función que no espera argumentos, lo cual está indicado por la lista vaciá `()`.

El primer programa en C

```
#include <stdio.h>          /* Incluye información acerca de la biblioteca
estándar */

main()                      /* Define una función llamada main que no recibe
valores de argumentos */
{                           /* Las proposiciones de main están encerradas entre
llaves */
  printf("Viva Peron!\n");  /* main llama a la función de biblioteca printf para
escribir esta secuencia de caracteres; \n representa el carácter nueva línea. */
}
```

Las *proposiciones* de una *función* están encerradas entre llaves `{…}`. La función `main` sólo contiene una proposición, `printf ("Viva Perón!\n");`. Una función se invoca al nombrarla, seguida de una lista de argumentos entre paréntesis; de esta manera se está llamando a la función `printf` con el argumento `"Viva Peron!\n"`. `printf` es una función de biblioteca que presenta la salida, en este caso la cadena de caracteres que se encuentra entre comillas. A una secuencia de caracteres entre comillas, como `"Viva Peron!\n"`, se le llama *cadena de caracteres* o *constante de cadena*. Por el momento, nuestro único uso de cadenas de caracteres será como argumentos para `printf` y otras funciones.

La secuencia `\n` en la cadena representa el carácter *nueva línea* en la notación de C, y hace avanzar la impresión al margen izquierdo de la siguiente línea. Si se omite el `\n` (un experimento que vale la pena hacer), encontrará que no hay avance de línea después de la impresión. Se debe utilizar `\n` para incluir un carácter de *nueva línea* en el argumento de `printf`; si se intenta algo como

```
printf("Viva Peron!
");
```

…el compilador de C producirá un mensaje de error.

`printf` nunca proporciona una nueva línea automáticamente, de manera que se pueden utilizar varias llamadas para construir una línea de salida en etapas.

Nuestro primer programa también pudo haber sido escrito de la siguiente manera:

```
#include <stdio.h>
main()
{
    printf ("Viva");
    printf ("Peron!");
    printf ("\n");
}
```

…produciéndose una salida idéntica.

Nótese que `\n` representa un solo carácter. Una secuencia de escape como `\n` proporciona un mecanismo general y extensible para representar caracteres invisibles o difíciles de escribir. Entre otros que C proporciona están `\t` para tabulación, `\b` para retroceso, `\"` para comillas, y `\\` para la barra diagonal invertida. Hay una lista completa en la sección 2.3.

- **Ejercicio 1-1**. Ejecute el programa "vivaperon" en su sistema. Experimente con la omisión de partes del programa, para ver qué mensajes de error se obtienen. □
- **Ejercicio 1-2**. Experimente el descubrir qué pasa cuando la cadena del argumento de `printf` contiene `\c`, en donde `c` es algún carácter no puesto en lista anteriormente. □

1.2 Variables y Expresiones Aritméticas

El siguiente programa utiliza la fórmula *°C =(5/9) x (°F-32)* para imprimir la siguiente tabla de temperaturas Fahrenheit y sus equivalentes centígrados o Celsius:

```
0       -17
20      -6
40      4
60      15
80      26
100     37
120     38
140     60
160     71
180     82
200     93
220     104
240     115
260     126
280     137
300     148
```

En si el programa aún consiste de la definición de una única función llamada `main`. Es más largo que el que imprime `Viva perón!`, pero no es complicado. Introduce varias ideas nuevas, incluyendo comentarios, declaraciones, variables, expresiones aritméticas, ciclos y salida con formato.

```
#include < std io .h >
/* imprime Tabla Fahrenheit-Celsius para fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int inferior, superior, paso;

    inferior = 0;      /* límite inferior de la tabla de temperaturas */
    superior = 300;    /* límite superior */
    paso = 20;         /* tamaño del incremento */

    fahr = inferior;
    while (fahr < = superior) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
```

```
        fahr = fahr + paso;
    }
}
```

Las dos líneas

```
/* imprime la tabla Fahrenheit-Celsius
        para fahr = 0, 20, ..., 300 */
```

...son un *comentario*, que en este caso explica brevemente lo que hace el programa. Cualesquier caracteres entre `/*` y `*/` son ignorados por el compilador, y pueden ser utilizados libremente para hacer a un programa más fácil de entender. Los comentarios pueden aparecer en cualquier lugar donde puede colocarse un espacio en blanco, un tabulador o nueva línea.

En C, se deben declarar todas las variables antes de su uso, generalmente al principio de la función y antes de cualquier proposición ejecutable. Una *declaración* notifica las propiedades de una variable; consta de un nombre de tipo y una lista de variables, como

```
int fahr, celsius;
int inferior, superior, paso;
```

El tipo `int` significa que las variables de la lista son enteros, en contraste con `float`, que significa punto flotante, esto es, números que pueden tener una parte fraccionaria. El rango tanto de `int` como de `float` depende de la máquina que se está utilizando; los `int` de 16 bits, que están comprendidos entre el `-32768` y `+32767`, son comunes, como lo son los `int` de 32 bits. Un número `float` típicamente es de 32 bits, por lo menos con seis dígitos significativos y una magnitud generalmente entre 10 a la -38 y 10 a la +38.

Además de `int` y `float`, C proporciona varios tipos de datos básicos, incluyendo:

`char`	carácter —un solo byte
`short`	entero corto
`long`	entero largo
`double`	punto flotante de doble precisión

Los tamaños de estos objetos también dependen de la máquina. También existen *arreglos*, *estructuras* y *uniones* de estos tipos básicos, *apuntadores* a ellos y *funciones* que regresan valores con esos tipos, todo lo cual se verá en el momento oportuno.

Los cálculos en el programa de conversión de temperaturas principian con las proposiciones de asignación.

```
inferior = 0 ;
superior = 300;
paso = 20 ;
```

…que asignan a las variables sus valores iniciales. Las proposiciones individuales se terminan con punto y coma `;`.

Cada línea de la tabla se calcula de la misma manera por lo que se utiliza una iteración que se repite una vez por cada línea de salida; este es el propósito del ciclo `while`:

```
while (fahr <= superior) {
    ...
}
```

El ciclo `while` funciona de la siguiente manera: se prueba la condición entre paréntesis. De ser verdadera (si `fahr` es menor o igual que `superior`), el cuerpo del ciclo (las tres proposiciones entre llaves) se ejecuta. Luego la condición se prueba nuevamente, y si es verdadera, el cuerpo se ejecuta de nuevo. Cuando la prueba resulta falsa (`fahr` excede a `superior`) la iteración termina, y la ejecución continúa en la proposición que sigue al ciclo. No existe ninguna otra proposición en este programa, de modo que termina.

El cuerpo de un `while` puede tener una o más proposiciones encerradas entre llaves, como en el convertidor de temperaturas, o una sola proposición sin llaves, como en:

```
while (i < j)
    i = 2 + i;
```

En cualquier caso, siempre se sangra la proposición controlada por el `while` con una tabulación (lo que se presenta aquí a cuatro espacios) para poder apreciar de un vistazo cuáles proposiciones están circunscriptas dentro del ciclo. El *sangrado* enfatiza la estructura lógica del programa. Aunque a los compiladores de C no les importa la apariencia del programa, un sangrado y espaciamiento adecuados son muy importantes para hacer programas fáciles de leer. Recomendamos escribir una sola proposición por línea y utilizar espacios en blanco alrededor de los operadores para dar claridad al agrupamiento. La posición de las llaves es menos importante, aunque los programadores del pueblo sostienen credos pasionales al respecto. Se eligió uno de los varios estilos populares. Escoja un estilo que le satisfaga y sea consistente en su uso.

La mayor parte del trabajo se realiza en el cuerpo del ciclo. La temperatura Celsius se calcula y se asigna a la variable celsius por la proposición.

```
celsius = 5 * (fahr–32) / 9;
```

La razón de multiplicar por 5 y después dividir entre 9 en lugar de solamente multiplicar por 5/9 es que en C - como en muchos otros lenguajes - la división de enteros trunca el resultado: cualquier parte fraccionaria se descarta. Puesto que 5 y 9 son enteros, 5/9 sería truncado a cero y así todas las temperaturas Celsius se reportarían como cero.

Este ejemplo también muestra un poco más acerca de cómo funciona `printf`. En realidad, `printf` es una función de propósito general para dar formato de salida, que se describirá con detalle en el capítulo 7. Su primer argumento es una cadena de caracteres que serán impresos, con cada `%` indicando en donde uno de los otros (segundo, tercero, ...) argumentos va a ser sustituido, y en qué forma será impreso. Por ejemplo, `%d` especifica un argumento entero, de modo que la proposición

```
printf("%d\t%d\n", fahr, celsius);
```

hace que los valores de los dos enteros `fahr` y `celsius` sean escritos, con una tabulación (`\t`) entre ellos.

Cada construcción `%` en el primer argumento de `printf` está asociada con el correspondiente segundo argumento, tercero, etc., y deben corresponder apropiadamente en número y tipo, o se tendrán soluciones incorrectas.

Con relación a esto, `printf` no es parte del lenguaje C; no existe propiamente una entrada o salida definida en C. `printf` es sólo una útil función de la biblioteca estándar de funciones que está accesible normalmente a los programas en C. Sin embargo, el comportamiento de `printf` está definido en el estándar ANSI, por lo que sus propiedades deben ser las mismas en cualquier compilador o biblioteca que se apegue a él.

Para concentrarnos en C, no hablaremos mucho acerca de la entrada y la salida hasta el capítulo 7. En particular, pospondremos el tema de la entrada con formato hasta entonces. Si se tiene que darle entrada a números, léase la discusión de la función `scanf` en la sección 7.4. La función `scanf` es como `printf`, exceptuando que lee de la entrada en lugar de escribir a la salida.

Existen un par de problemas con el programa de conversión de temperaturas.

El más simple es que la salida no es muy estética debido a que los números no están justificados hacia su derecha. Esto es fácil de corregir; si aumentamos a cada `%d` de la proposición `printf` una amplitud, los números impresos serán justificados hacia su derecha dentro de sus campos. Por ejemplo, podría decirse:

```
printf("%3d %6d\n", fahr, celsius);
```

para escribir el primer número de cada línea en un campo de tres dígitos de ancho, y el segundo en un campo de seis dígitos, como esto:

```
0    -17
20    -6
40     4
60    15
80    26
100   37
...
```

El problema más grave es que debido a que se ha utilizado aritmética de enteros, las temperaturas Celsius no son muy precisas; por ejemplo, 0ºF es en realidad aproximadamente —17.8°C, no —17ºC. Para obtener soluciones más precisas, se debe utilizar aritmética de punto flotante en lugar de entera. Esto requiere de algunos cambios en el programa. Aquí está una segunda versión:

```
#include <stdio.h>

/* imprime la tabla Fahrenheit-Celsius
   para fahr = 0, 20, ..., 300; versión de coma flotante */
main()
{
    float fahr, celsius;
    float inferior, superior, paso;

    inferior = 0;     /* limite inferior de escala de temperatura */
    superior = 300;   /* limite superior */
    paso = 20;        /* tamaño de paso */

    fahr = inferior;
    while (fahr <= superior) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + paso;
    }
}
```

Esto es muy semejante a lo anterior, excepto que `fahr` y `celsius` están declarados como `float`, y la fórmula de conversión está escrita en una forma más natural. No pudimos utilizar `5/9` en la versión anterior debido a que la división entera lo truncaría a cero. Sin embargo, un punto decimal en una constante indica que ésta es de punto flotante, por lo que `5.0/9.0` no se trunca debido a que es una relación de dos valores de punto flotante.

Si un operador aritmético tiene operandos enteros, se ejecuta una operación entera. Si un operador numérico tiene un operando de punto flotante y otro entero, este último será convertido a punto flotante antes de hacer la operación. Si se hubiera escrito `(fahr-32)`, el 32 sería convertido automáticamente a punto flotante. Escribir constantes de punto flotante con puntos decimales explícitos, aun cuando tengan valores enteros, destaca su naturaleza de punto flotante para los lectores humanos.

Las reglas detalladas de cuándo los enteros se convierten a punto flotante se encuentran en el capítulo 2. Por ahora, nótese que la asignación

```
fahr = inferior;
```

y la prueba

```
while (fahr <= superior)
```

también trabajan en la forma natural (el `int` se convierte a `float` antes de efectuarse la operación).

La especificación de conversión `%3.0f` del `printf` indica que se escribirá un número de punto flotante (en este caso `fahr`) por lo menos con tres caracteres de ancho, sin punto decimal y sin dígitos fraccionarios; `%6.1f` describe a otro número (`celsius`) que se escribirá en una amplitud de por lo menos 6 caracteres, con 1 dígito después del punto decimal. La salida tendrá el siguiente aspecto:

```
 0      -17.8
20       -6.7
40        4.4
...
```

La amplitud y la precisión pueden omitirse de una especificación: `%6f` indica que el número es por lo menos de seis caracteres de ancho; `%.2f` indica dos caracteres después del punto decimal, pero el ancho no está restringido; y `%f` únicamente indica escribir el número como punto flotante.

`%d`	Escribe como entero decimal
`%6d`	escribe como entero decimal, por lo menos con 6 caracteres de amplitud.
`%f`	escribe como punto flotante
`%6f`	escribe como punto flotante, por lo menos con 6 caracteres de amplitud
`%.2f`	escribe como punto flotante, con 2 caracteres después del punto decimal.
`%6.2f`	escribe como punto flotante, por lo menos con 6 caracteres, y con dos caracteres después del punto decimal.

Entre otros, `printf` también reconoce `%o` para octal, `%x` para hexadecimal, `%c` para carácter, `%s` para cadena de caracteres y `%%` para denotar el caracter `%` en sí.

- **Ejercicio 1-3**. Modifique el programa de conversión de temperaturas de modo que escriba un encabezado sobre la tabla. □
- **Ejercicio 1-4**. Escriba un programa que imprima la tabla correspondiente Celsius a Fahrenheit. □

1.3 La proposición for

Existen suficientes formas distintas de escribir un programa para una tarea en particular. Intentemos una variación del programa de conversión detem peraturas.

```
#include <stdio.h>

/* imprime la tabla Fahrenheit-Celsius */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Este produce los mismos resultados, pero ciertamente se ve diferente. Un cambio importante es la eliminación de la mayoría de las variables; sólo permanece `fahr` y la hemos convertido en `int`. Los límites inferior y superior y el tamaño del paso sólo aparecen como constantes dentro de la proposición `for`, que es una nueva construcción, y la expresión que calcula la temperatura Celsius ahora aparece como el tercer argumento de `printf` en vez de una proposición de asignación separada.

Este último cambio ejemplifica una regla general — en cualquier contexto en el que se permita utilizar el valor de una variable de algún tipo, es posible usar una expresión más complicada de ese tipo. Puesto que el tercer argumento de `printf` debe ser un valor de punto flotante para coincidir con `%6.1f`, cualquier expresión de punto flotante puede ocurrir aquí.

La proposición `for` es un ciclo, una forma generalizada del `while`. Si se compara con el `while` anterior, su operación debe ser clara. Dentro de los paréntesis existen tres secciones, separadas por punto y coma `;`. La primera, la inicialización

```
fahr = 0
```

se ejecuta una vez, antes de entrar propiamente al ciclo. La segunda sección es la condición o prueba que controla el ciclo:

```
fahr <= 300
```

Esta condición se evalúa; si es verdadera, el cuerpo del ciclo (en este caso un simple `printf`) se ejecuta. Después el incremento de avance

```
fahr = fahr + 20
```

se ejecuta y la condición se vuelve a evaluar. El ciclo termina si la condición se hace falsa. Tal como con el `while`, el cuerpo del ciclo puede ser una proposición sencilla o un grupo de proposiciones encerradas entre llaves. La inicialización, la condición y el incremento pueden ser cualquier expresión.

La selección entre `while` y `for` es arbitraria, y se basa en aquello que parezca más claro. El `for` es por lo general apropiado para ciclos en los que la inicialización y el incremento son proposiciones sencillas y lógicamente relacionadas, puesto que es más compacto que el `while` y mantiene reunidas en un lugar a las proposiciones que controlan al ciclo.

- **Ejercicio 1-5**. Modifique el programa de conversión de temperaturas de manera que escriba la tabla en orden inverso, esto es, desde 300 grados hasta 0. □

1.4 Constantes simbólicas

Una observación final antes de dejar definitivamente el tema de la conversión de temperaturas. Es una mala práctica poner “números mágicos” tales como `300` y `20` en un programa, ya que le proporcionan muy poca información a

quien tenga que leer el programa, y son difíciles de modificar de forma sistemática. Una manera de tratar a esos números mágicos es darles nombres significativos. Una línea `#define` define un *nombre simbólico* o *constante simbólica* como una cadena de caracteres particularmente especial:

```
#define nombre  texto de reemplazo
```

A partir de esto, cualquier ocurrencia de *nombre* (que no esté entre comillas ni como parte de otro nombre) se sustituirá por el *texto de reemplazo* correspondiente. El *nombre* tiene la misma forma que un nombre de variable: una secuencia de letras y dígitos que comienza con una letra. El *texto de reemplazo* puede ser cualquier secuencia de caracteres; no está limitado a números.

```
#include <stdio.h>
#define INFERIOR   0     /* límite inferior de la tabla »/
#define SUPERIOR    300  /* limite superior */
#define PASO        20   /* tamaño del paso de incremento */

/* imprime la tabla Fahrenheit-Celsius */
main()
{
    int fahr;

    for (fahr = INFERIOR; fahr <= SUPERIOR; fahr = fahr + PASO)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Las cantidades `INFERIOR`, `SUPERIOR` y `PASO` son constantes simbólicas, no variables, por lo que no aparecen entre las declaraciones. Por convención, los nombres de constantes simbólicas, se escriben con letras mayúsculas, de forma tal que puedan distinguirse fácilmente de los nombres de variables (escritos con minúsculas). Nótese que no hay punto y coma al final de una línea `#define`.

1.5 Entrada y salida de caracteres

Ahora vamos a considerar una familia de programas relacionados para el procesamiento de datos de tipo carácter. Se encontrará que muchos programas sólo son versiones ampliadas de los prototipos que se tratan aquí.

El modelo de entrada y salida manejado por la biblioteca estándar es muy simple. La entrada y salida de texto, sin importar dónde fue originada o hacia dónde se dirige, se tratan como flujos (“streams”) de caracteres. Un *flujo de texto* es una secuencia de caracteres divididos entre líneas, cada una de las cuales consta de cero o más caracteres seguidos de un carácter *nueva línea*. La biblioteca es responsable de hacer que cada secuencia de entrada o salida esté de acuerdo con este modelo; el programador de C que utiliza la biblioteca no necesita preocuparse de cómo están representadas las líneas fuera del programa.

La biblioteca estándar proporciona varias funciones para leer o escribir de a un carácter a la vez, de las cuales `getchar` y `putchar` son las más simples. Cada vez que es invocada, `getchar` lee *el siguiente carácter de entrada* de una secuencia de texto y lo devuelve como su valor. Esto es, después de

```
c = getchar()
```

la variable `c` contiene el siguiente carácter de entrada. Los caracteres provienen normalmente del teclado; la entrada de archivos se trata en el capítulo 7.

La función `putchar` escribe un carácter cada vez que se invoca:

```
putchar(c);
```

escribe el contenido de la variable entera `c` como un carácter, generalmente en la pantalla; Las llamadas a `putchar` y a `printf` pueden estar alternadas; la salida aparecerá en el orden en que se realicen las llamadas.

1.5.1 Copia de archivos

Con `getchar` y `putchar` se puede escribir una cantidad sorprendente de código útil sin saber nada más acerca de entrada y salida. El ejemplo más sencillo es un programa que copia la entrada en la salida, un carácter a la vez:

```
lee un carácter
while (carácter no es indicador de fin de archivo)
        manda a la salida el carácter recién leído
        lee un carácter
```

Al convertir esto en C se obtiene

```
#include <stdio.h>
/* copia la entrada a la salida; la. versión */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

El operador de relación `!=` significa “no igual a”.

Lo que aparece como un carácter en el teclado o en la pantalla es, por supuesto, como cualquier otra cosa, almacenado internamente como un patrón de bits. El tipo `char` tiene la función específica de almacenar ese tipo de dato, pero también puede ser usado cualquier tipo de entero. Usamos `int` por una sutil pero importante razón.

El problema es distinguir el fin de la entrada de los datos válidos. La solución es que getchar devuelve un valor distintivo cuando no hay más a la entrada,

un valor que no puede ser confundido con ningún otro carácter. Este valor se llama , por “end-of-file” (“fin de archivo”). Se debe declarar `c` con un tipo que sea lo suficientemente grande para almacenar cualquier valor que le regrese `getchar`. No se puede utilizar `char` puesto que `c` debe ser suficientemente grande como para mantener a además de cualquier otro carácter. Por lo tanto, se emplea `int`.

es un entero definido en `<stdio.h>`, pero el valor numérico específico no importa mientras que no sea el mismo que ningún valor tipo `char`. Utilizando la constante simbólica, hemos asegurado que nada en el programa depende del valor numérico específico.

El programa para copiar podría escribirse de modo más conciso por programadores experimentados de C. En lenguaje C, cualquier asignación, tal como

```
c = getchar();
```

es una expresión y tiene un valor: el del lado izquierdo luego de la asignación. Esto significa que una asignación puede aparecer como parte de una expresión más larga. Si la asignación de un carácter a c se coloca dentro de la sección de prueba de un ciclo while, el programa que copia puede escribirse de la siguiente manera:

```
#include <stdio.h>
/* copia la entrada a la salida; 2a. versión */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

El `while` obtiene un carácter, lo asigna a `c`, y entonces prueba si el carácter fue la señal de fin de archivo. De no serlo, el cuerpo del `while` se ejecuta, escribiendo el carácter; luego se repite el `while`. Luego, cuando se alcanza el final de la entrada, el `while` termina y también lo hace `main`.

Esta versión centraliza la entrada — ahora hay sólo una referencia a `getchar` — y reduce el programa. El programa resultante es más compacto y más fácil de leer una vez que se domina el truco. Usted verá seguido este estilo. (Sin embargo, es posible descarriarse y crear código impenetrable, una tendencia que trataremos de reprimir.)

Los paréntesis que están alrededor de la asignación dentro de la condición son necesarios. La *precedencia* de `!=` es más alta que la de `=`, lo que significa que en ausencia de paréntesis la prueba de relación `!=` se realizaría antes de la asignación `=`. De esta manera, la proposición

```
c = getchar() != EOF
```

es equivalente a

```
c = (getchar() != EOF)
```

Esto tiene el efecto indeseable de hacer que `c` sea `0` o `1`, dependiendo de si la llamada de `getchar` encontró fin de archivo. (En el capítulo 2 se trata este tema con más detalle).

- **Ejercicio 1-6**. Verifique que la expresión `getchar() !=` es `0` o `1`. □
- **Ejercicio 1-7**. Escriba un programa que imprima el valor de . □

1.5.2 Conteo de caracteres

El siguiente programa cuenta caracteres y es semejante al programa que copia.

```
#include <stdio.h>
/* cuenta los caracteres de la entrada; la. versión */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

La proposición

```
++nc;
```

presenta un nuevo operador, ++, que significa incrementa en uno. Es posible escribir `nc = nc + 1`, pero `++nc` es más conciso y muchas veces más eficiente. Hay un operador correspondiente `--` para disminuir en `1`. Los operadores `++` y `--` pueden ser tanto operadores *prefijos* (`++nc`) como *postfijos* (`nc++`); esas dos formas tienen diferentes valores dentro de las expresiones, como se demostrará en el capítulo 2, pero ambos `++nc` y `nc++` incrementan a `nc`. Por el momento adoptaremos la forma de *prefijo*.

El programa para contar caracteres acumula su cuenta en una variable `long` en lugar de una `int`. Los enteros `long` son por lo menos de 32 bits. Aunque en algunas máquinas *int* y *long* son del mismo tamaño, en otras un *int* es de 16 bits, con un valor máximo de `32767`, y tomaría relativamente poca lectura a la entrada para desbordar un contador *int*. La especificación de conversión `%ld` indica a `printf` que el argumento correspondiente es un entero *long*.

Sería posible tener la capacidad de trabajar con números mayores empleando un `double` (*float* de doble precisión). También se utilizará una proposición `for` en lugar de un `while`, para demostrar otra forma de escribir el ciclo.

```
#include <stdio.h>
/* cuenta los caracteres de la entrada; 2a. versión */
main()
{
```

```
    double nc;
    for (nc = 0; gechar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

`printf` utiliza `%f` tanto para *float* como para *double*; `%.0f` suprime la impresión del punto decimal y de la parte fraccionaria, que es cero.

El cuerpo de este ciclo `for` está vacío, debido a que todo el trabajo se realiza en las secciones de prueba e incremento. Pero las reglas gramaticales de C requieren que una proposición `for` tenga un cuerpo. El `;` aislado se llama *proposición nula*, y está aquí para satisfacer este requisito. Lo colocamos en una línea aparte para que sea visible.

Antes de abandonar el programa para contar caracteres, obsérvese que si la entrada no contiene caracteres, la prueba del `while` o del `for` no tiene éxito desde la primera llamada `getchar`, y el programa produce cero, el resultado correcto. Esto es importante. Uno de los aspectos agradables acerca del `while` y del `for` es que hacen la prueba al inicio del ciclo, antes de proceder con el cuerpo. Si no hay nada que hacer, nada se hace, aun si ello significa no pasar a través del cuerpo del ciclo. Los programas deben actuar en forma inteligente cuando se les da una entrada de longitud cero. Las proposiciones `while` y `for` ayudan a asegurar que los programas realizan cosas razonables con condiciones de frontera.

1.5.3 Conteo de líneas

El siguiente programa cuenta líneas a la entrada. Como se mencionó anteriormente, la biblioteca estándar asegura que una secuencia de texto de entrada parezca una secuencia de líneas, cada una terminada por un carácter *nueva línea*.

Por lo tanto, contar líneas es solamente contar caracteres *nueva línea*:

```
#include <stdio.h>
/* cuenta las líneas de la entrada */
main()
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

El cuerpo del `while` consiste ahora en un `if`, el cual a su vez controla el incremento `++nl`. La proposición `if` prueba la condición que se encuentra entre paréntesis y, si la condición es verdadera, ejecuta la proposición (o grupo de proposiciones entre llaves) que le sigue. Hemos sangrado nuevamente para mostrar lo que controla cada elemento.

El doble signo de igualdad == es la notación de C para expresar “igual a” (como el = simple de Pascal o el `.EQ.` de Fortran). Este símbolo se emplea para distinguir la prueba de igualdad del = simple que utiliza C para la asignación. Un mensaje de alerta: los principiantes de C ocasionalmente escriben = cuando en realidad deben usar == . Como se verá en el capítulo 2, el resultado es por lo general una expresión legal, de modo que no se obtendrá ninguna advertencia.

Un carácter escrito entre apóstrofos `'...'` representa un valor entero igual al valor numérico del carácter en el conjunto de caracteres de la máquina. Esto se llama una *constante de carácter*, aunque sólo es otra forma de escribir un pequeño entero. Así, por ejemplo `'A'` es una constante de carácter; en el conjunto de caracteres su valor es `65` (esta es la representación interna del carácter `A`). Por supuesto `'A'` es preferible que `65`: su significado es obvio, y es independiente de un conjunto de caracteres en particular.

Las secuencias de escape que se utilizan en constantes de cadena también son legales en constantes de carácter; así, `'\n'` significa el valor del carácter *nueva línea*, el cual es 10 del código . Se debe notar cuidadosamente que `'\n'` es un carácter simple, y en expresiones es sólo un entero; por otro lado, `'\n'` es una constante de cadena que contiene sólo un carácter. En el capítulo 2 se trata el tema de cadenas versus caracteres.

- **Ejercicio 1-8**. Escriba un programa que cuente espacios en blanco, tabuladores y nuevas líneas. □
- **Ejercicio 1-9**. Escriba un programa que copie su entrada a la salida, reemplazando cada cadena de uno o más blancos por un solo blanco. □
- **Ejercicio 1-10**. Escriba un programa que copie su entrada a la salida, reemplazando cada tabulación por `\t`, cada retroceso por `\b` y cada diagonal invertida por `\\`. Esto hace que las tabulaciones y los espacios sean visibles sin confusiones. □

1.5.4 Conteo de palabras

El cuarto en nuestra serie de programas útiles cuenta las líneas, palabras y caracteres, usando la definición de que una palabra es cualquier secuencia de caracteres que no contiene espacio en blanco ni tabulación ni nueva línea. Esta es una versión reducida del programa *wc* de UNIX.

```
#include <stdio.h>

#define DENTRO  1 /* en una palabra */
#define FUERA   0 /* fuera de una palabra */

/* cuenta lineas, palabras, y caracteres de la entrada */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
```

```
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c = '\t')
            state = FUERA;
        else if (state == FUERA) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Cada vez que el programa encuentra el primer carácter de una palabra, contabiliza una palabra más. La variable `state` registra si actualmente el programa está o no sobre una palabra; al iniciar es “no está sobre una palabra”, por lo que se asigna el valor `DENTRO`. Es preferible usar las constantes simbólicas `DENTRO` y `FUERA` que los valores literales `1` y `0`, porque hacen el programa más legible. En un programa tan pequeño como éste, la diferencia es mínima, pero en programas más grandes el incremento en claridad bien vale el esfuerzo extra que se haya realizado para escribir de esta manera desde el principio. También se descubrirá que es más fácil hacer cambios extensivos en programas donde los números mágicos aparecen sólo como constantes simbólicas.

La línea

```
n1 = nw = nc = 0;
```

inicializa a las tres variables en cero. Este no es un caso especial sino una consecuencia del hecho de que una asignación es una expresión con un valor, y que las asignaciones se asocian de derecha a izquierda. Es como si se hubiese escrito

```
n1 = (nw = (nc = 0));
```

El operador `||` significa “OR”, por lo que la línea

```
if (c == ' ' || c == '\n' || c = '\t')
```

dice “si `c` es un *caracter en blanco* o `c` es *nueva línea*, o `c` es un *tabulador*”. (Recuerde que la secuencia de escape `\t` es una representación visible del carácter *tabulador*). Existe un correspondiente operador && para *AND*; su precedencia es más alta que la de `||`. Las expresiones conectadas por && o `||` se evalúan de izquierda a derecha, y se garantiza que la evaluación terminará tan pronto como se conozca la verdad o falsedad. Si `c` es un *caracter en blanco*, no hay necesidad de probar si es una nueva línea o un tabulador, de modo que esas pruebas no se hacen. Esto no es de particular importancia en este caso, pero es significativo en situaciones más complicadas, como se verá más adelante.

El ejemplo muestra también un `else`, el cual especifica una acción alternativa si la condición de una proposición `if` es falsa. La forma general es

```
if (expresión)
    proposición1
else
    proposición2
```

Una y sólo una de las dos proposiciones asociadas con un `if-else` se realiza. Si la *expresión* es verdadera, se ejecuta *proposición1* si no lo es, se ejecuta *proposición2*. Cada *proposición* puede ser una proposición sencilla o varias entre llaves. En el programa para contar palabras, la que está después del `else` es un `if` que controla dos proposiciones entre llaves.

- **Ejercicio 1-11**. ¿Cómo probaría el programa para contar palabras? ¿Qué clase de entrada es la más conveniente para descubrir errores si éstos existen? □
- **Ejercicio 1-12**. Escriba un programa que imprima su entrada una palabra por línea. □

1.6 Arreglos

Escribamos un programa para contar el número de ocurrencias de cada dígito, de caracteres espaciadores (caracter en blancos, tabuladores, nueva línea), y de todos los otros caracteres. Esto es artificioso, pero nos permite ilustrar varios aspectos de C en un programa.

Existen doce categorías de entrada, por lo que es conveniente utilizar un arreglo para mantener el número de ocurrencias de cada dígito, en lugar de tener diez variables individuales. Esta es una versión del programa:

```
#include <stdio.h>

/* cuenta dígitos, espacios blancos, y otros */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;
    printf("digitos =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", espacio en blanco = %d, otro = %d\n",
        nwhite, nother);
}
```

La salida de este programa al ejecutarlo sobre sí mismo es

```
dígitos = 9 3 0 0 0 0 0 0 0 1, espacios en blancos = 123, otros = 345
```

La declaración

```
int ndigit [ 1 0 ];
```

declara `ndigit` como un arreglo de 10 enteros. En C, los subíndices de arreglos comienzan en cero, por lo que los elementos son `ndigit[0]`, `ndigit[1]`, `ndigit[9]`. Esto se refleja en los ciclos `for` que inicializan e imprimen el arreglo.

Un subíndice puede ser cualquier expresión entera, lo que incluye a variables enteras como `i`, y constantes enteras.

Este programa en particular se basa en las propiedades de la representación de los dígitos como caracteres. Por ejemplo, la prueba

```
if (c >= '0' && c <= '9')
```

determina si el carácter en `c` es un dígito. Si lo es, el valor numérico del dígito es

```
c - '0'
```

Esto sólo funciona si '0', '1', ..., '9' tienen valores consecutivos ascendentes. Por fortuna, esto es así en todos los conjuntos de caracteres.

Por definición, los `char` son sólo pequeños enteros, por lo que las variables y las constantes `char` son idénticas a las `int` en expresiones aritméticas. Esto es natural y conveniente; por ejemplo, `c - '0'` es una expresión entera con un valor entre 0 y 9, correspondiente a los caracteres ''0'' a ''9'' almacenados en `c`, por lo que es un subíndice válido para el arreglo `ndigit`.

La decisión de si un carácter es dígito, espacio en blanco u otra cosa se realiza con la secuencia

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

El patrón

```
if (condición 1)
        proposición 1
else if (condición 2)
        proposición2
        ...
        ...
else
        proposición_n,
```

se encuentra frecuentemente en programas como una forma de expresar una decisión múltiple. Las *condiciones* se evalúan en orden desde el principio hasta que se satisface alguna *condición*; en ese punto se ejecuta la *proposición* correspondiente, y la construcción completa termina. (Cualquier *proposición* puede constar de varias proposiciones entre llaves.) Si no se satisface ninguna de las condiciones, se ejecuta la *proposición que está después* del `else` final (si es que esta existe). Cuando se omiten el `else` y la *proposición* finales, tal como se hizo en el programa para contar palabras, no se lleva a cabo ninguna acción. Puede haber cualquier número de grupos de

```
else if (condición)
    proposición
```

entre el `if` inicial y el `else` final.

Se recomienda, por estilo, escribir esta construcción tal como se ha mostrado; si cada `if` estuviese sangrado después del `else` anterior, una larga secuencia de decisiones podría rebasar el margen derecho de la página.

La proposición `switch`, que se tratará en el capítulo 3, proporciona otra forma de escribir una decisión múltiple, que es particularmente apropiada cuando la condición es determinar si alguna expresión entera o de carácter corresponde con algún miembro de un conjunto de constantes. Para contrastar, se presentará una versión de este programa, usando `switch`, en la sección 3.4.

- **Ejercicio 1-13**. Escriba un programa que imprima el histograma de las longitudes de las palabras de su entrada. Es fácil dibujar el histograma con las barras horizontales; la orientación vertical es un reto más interesante. □
- **Ejercicio 1-14**. Escriba un programa que imprima el histograma de las frecuencias con que se presentan diferentes caracteres leídos a la entrada. □

1.7 Funciones

En lenguaje C, una función es el equivalente a una subrutina o función en Fortran, o a un procedimiento o función en Pascal. Una función proporciona una forma conveniente de encapsular algunos cálculos, que se pueden emplear después sin preocuparse de su implantación. Con funciones diseñadas adecuadamente, es posible ignorar *cómo* se realiza un trabajo; basta con saber *qué* hace. El lenguaje C hace que el uso de funciones sea fácil, conveniente y eficiente; es común ver una función corta definida y empleada una sola vez, únicamente porque eso esclarece alguna parte del código.

Hasta ahora sólo se han utilizado funciones como `printf`, `getchar` y `putchar`, que nos han sido proporcionadas; ya es el momento de escribir unas pocas nosotros mismos. Dado que C no posee un operador de exponenciación como el ** de Fortran, ilustremos el mecanismo de la definición de una función al escribir la función `power(m,n)`, que eleva un entero `m` a una potencia entera y

positiva n. Esto es, el valor de power(2,5) es 32. Esta función no es una rutina de exponenciación práctica, puesto que sólo maneja potencias positivas de enteros pequeños, pero es suficiente para ilustración (la biblioteca estándar contiene una función pow(x,y) que calcula x^y).

A continuación se presenta la función power y un programa main para utilizarla, de modo que se vea la estructura completa de una vez.

```
#include <stdio.h>

int power(int m, int n);

/* prueba la función power */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power: eleva la base a la n-ésima potencia; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Una definición de función tiene la forma siguiente:

```
tipo-de-retorno nombre-de-función (declaración de parámetros, si los hay)
{
    declaraciones
    proposiciones
}
```

Las definiciones de función pueden aparecer en cualquier orden y en uno o varios archivos fuente, pero una función no puede separarse en archivos diferentes. Si el programa fuente aparece en varios archivos, tal vez se tengan que especificar más cosas al compilar y cargarlo que si estuviera en uno solo, pero eso es cosa del sistema operativo, no un atributo del lenguaje. Por ahora supondremos que ambas funciones están en el mismo archivo y cualquier cosa que se haya aprendido acerca de cómo ejecutar programas en C, aún funcionarán.

La función power es invocada dos veces por main, en la línea

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Cada llamada pasa dos argumentos a power, que cada vez regresa un entero, al que se pone formato y se imprime. En una expresión, power(2,i) es un

entero tal como lo son 2 e i. (No todas las funciones producen un valor entero; lo que se verá en el capítulo 4).

La primera línea de la función power,

```
int power(int base, int n)
```

declara los tipos y nombres de los parámetros, así como el tipo de resultado que la función devuelve. Los nombres que emplea power para sus parámetros son locales a la función y son invisibles a cualquier otra función: otras rutinas pueden utilizar los mismos nombres sin que exista problema alguno. Esto también es cierto para las variables i y p: la i de power no tiene nada que ver con la i de main.

Generalmente usaremos parámetro para una variable nombrada en la lista entre paréntesis de la definición de una función, y argumento para el valor empleado al hacer la llamada de la función. Los términos argumento formal y argumento real se emplean en ocasiones para hacer la misma distinción.

El valor que calcula power se regresa a main por medio de la proposición return, a la cual le puede seguir cualquier expresión:

```
return expresión;
```

Una función no necesita regresar un valor; una proposición return sin expresión hace que el control regrese al programa, pero no devuelve algún valor de utilidad, como se haría al “caer al final” de una función al alcanzar el caracter } de *llave cerrada* que opera de terminación. Además, la función que llama puede ignorar el valor que regresa una función.

Probablemente haya notado que hay una proposición return al final de main. Puesto que main es una función como cualquier otra, también puede regresar un valor a quien la invoca, que es en efecto el medio ambiente en el que el programa se ejecuta. Típicamente, un valor de regreso cero implica una terminación normal; los valores diferentes de cero indican condiciones de terminación no comunes o erróneas. En busca de la simplicidad, se han omitido hasta ahora las proposiciones return de las funciones main, pero se incluirán más adelante, como un recordatorio de que los programas deben regresar su estado final a su medio ambiente.

La declaración

```
int power(int m, int n);
```

precisamente antes de main, indica que power es una función que espera dos argumentos int y regresa un int. Esta declaración, a la cual se le llama *función prototipo*, debe coincidir con la definición y uso de power. Es un error el que la definición de una función o cualquier uso que de ella se haga no corresponda con su prototipo.

Los nombres de los parámetros no necesitan coincidir; de hecho, son opcionales en el prototipo de una función, de modo que para el prototipo se pudo haber escrito

```
int power(int, int);
```

No obstante, unos nombres bien seleccionados son una buena documentación, por lo que se emplearán frecuentemente.

Una nota histórica: La mayor modificación entre ANSI C y las versiones anteriores es cómo están declaradas y definidas las funciones. En la definición original de C, la función `power` se pudo haber escrito de la siguiente manera:

```
/* power: eleva la base a n-ésima potencia; n >= 0 */
/*        (versión en estilo antiguo) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1 ;
    for (i = 1 ; i < = n; + + i)
        p — p * base;
    return p;
}
```

Los parámetros se nombran entre los paréntesis y sus tipos se declaran antes del caracter { de llave abierta; los parámetros que no se declaran se toman como `int`. {El cuerpo de la función es igual a la anterior).

La declaración de `power` al inicio del programa pudo haberse visto como sigue:

```
int power();
```

No se permitió ninguna lista de parámetros, de modo que el compilador no pudo revisar con facilidad que `power` fuera llamada correctamente. De hecho, puesto que por omisión se podía suponer que `power` regresaba un entero `int`, toda la declaración podría haberse omitido.

La nueva sintaxis de los prototipos de funciones permite que sea mucho más fácil para el compilador detectar errores en el número o tipo de argumentos. El viejo estilo de declaración y definición aún funciona en ANSI C, al menos por un periodo de transición, pero se recomienda ampliamente utilizar la nueva forma si se tiene un compilador que le de soporte.

- **Ejercicio 1-15**. Escriba de nuevo el programa de conversión de temperatura de la sección 1.2, de modo que utilice una función para la conversión.

1.8 Argumentos -- llamadas por valor

Hay un aspecto de las funciones de C que puede parecer poco familiar a los programadores acostumbrados a otros lenguajes, particularmente Fortran. En

C, todos los argumentos de una función se pasan “por valor” . Esto significa que la función que se invoca recibe los valores de sus argumentos en variables temporales y no en las originales. Esto conduce a algunas propiedades diferentes a las que se ven en lenguajes con “llamadas por referencia” como Fortran o con parámetros *var* en Pascal, en donde la rutina que se invoca tiene acceso al argumento original, no a una copia local.

> La diferencia principal es que en C la función que se invoca no puede alterar directamente una variable de la función que hace la llamada; sólo puede modificar su copia privada y temporal.

Sin embargo, la llamada por valor es una ventaja, no una desventaja. Por lo general, esto conduce a elaborar programas más compactos con pocas variables extrañas, puesto que los parámetros se tratan en la función invocada como variables locales convenientemente inicializadas. Por ejemplo, he aquí una versión de `power` que utiliza esta propiedad.

```
/* power: eleva la base a la n-ésima potencia; n > = 0 ; versión 2 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

El parámetro `n` se utiliza como una variable temporal, y se decrementa (un ciclo `for` que se ejecuta hacia atrás) hasta que llega a cero; ya no es necesaria la variable `i`. Cualquier cosa que se le haga a `n` dentro de `power` no tiene efecto sobre el argumento con el que se llamó originalmente `power`.

Cuando sea necesario, es posible hacer que una función modifique una variable dentro de una rutina invocada. La función que llama debe proporcionar la *dirección* de la variable que será cambiada (técnicamente un *apuntador* a la variable), y la función que se invoca debe declarar que el parámetro sea un apuntador y tenga acceso a la variable indirectamente a través de él. Los apuntadores se tratarán en el capítulo 5.

La historia es diferente con los arreglos. Cuando el nombre de un arreglo se emplea como argumento, el valor que se pasa a la función es la localización o la dirección del principio del arreglo — no hay copia de los elementos del arreglo. Al colocarle subíndices a este valor, la función puede tener acceso y alterar cualquier elemento del arreglo. Este es el tema de la siguiente sección.

1.9 Arreglos de caracteres

El tipo de arreglo más común en C es el de caracteres. Para ilustrar el uso de arreglos de caracteres y funciones que los manipulan, escriba un programa que lea un conjunto de líneas de texto e imprima la de mayor longitud. El pseudocódigo es bastante simple:

```
while (hay otra línea)
    if (es más larga que la anterior más larga)
        guárdala
        guarda su longitud
imprime la línea más larga
```

Este pseudocódigo deja en claro que el programa se divide naturalmente en partes. Una trae una nueva línea, o trae la prueba y el resto controla el proceso.

Puesto que la división de las partes es muy fina, lo correcto será escribirlas de ese modo. Así pues, escribamos primero un a función `getline` para extraer la siguiente línea de la entrada. Trataremos de hacer a la función útil en otros contextos. Al menos, `getline` tiene que regresar una señal acerca de la posibilidad de un fin de archivo; un diseño de más utilidad deberá retornar la longitud de la línea, o cero si se encuentra el fin de archivo. Cero es un regreso de fin de archivo aceptable debido a que nunca es una longitud de línea válida. Cada línea de texto tiene al menos un carácter; incluso una línea que sólo contenga un carácter nueva línea, tiene longitud 1.

Cuando se encuentre una línea que es mayor que la anteriormente más larga, se debe guardar en algún lugar. Esto sugiere una segunda función `copy`, para copiar la nueva línea a un lugar seguro.

Finalmente, se necesita un programa principal para controlar `getline` y `copy`.

El resultado es el siguiente:

```
#include <stdio.h>
#define MAXLINE 1000    /* tamaño máximo de la línea de entrada */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* imprime la línea de entrada más larga */
main()
{
    int len;               /* longitud actual de la línea */
    int max;               /* máxima longitud vista hasta el momento */
    char line[MAXLINE];    /* línea de entrada actual */
    char longest[MAXLINE]; /* la línea más larga se guarda aquí */

    max = 0 ;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
          max = len;
          copy(longest, line);
        }
    if (max > 0 )    /* hubo una línea */
        printf("%s", longest);
    return 0 ;
}

/* getline: lee una línea en s, regresa su longitud */
int getline(char s[], int lim)
{
    int c, i;
```

```
    for (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
    }
    s[i] = '\0';
    return i;
}
/* copy: copia 'from' en 'to'; supone que to es suficientemente grande */
void copy(char to[], char from[])
{
    int i;

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Las funciones `getline` y `copy` están declaradas al principio del programa, que se supone está contenido en un archivo. `main` y `getline` se comunican a través de un par de argumentos y un valor de retorno. En `getline` los argumentos se declaran por la línea

```
int getline(char s[], int lim);
```

que especifica que el primer argumento, `s`, es un arreglo, y el segundo, `lim`, es un entero. El propósito de proporcionar el tamaño de un arreglo es fijar espacio de almacenamiento contiguo. La longitud del arreglo `s` no es necesaria en `getline`, puesto que su tamaño se fija en `main`. En `getline` se utiliza `return` para regresar un valor a quién lo llama, tal como hizo la función `power`. Esta línea también declara que `getline` regresa un int; puesto que int es el valor de retorno por omisión, puede suprimirse.

Algunas funciones regresan un valor útil; otras, como `copy`, se emplean únicamente por su efecto y no regresan un valor. El tipo de retorno de `copy` es void, el cual establece explícitamente que ningún valor se regresa.

En `getline` se coloca el carácter `\0` (*carácter nulo*, cuyo valor es cero) al final del arreglo que está creando, para marcar el fin de la cadena de caracteres. Esta convención también se utiliza por el lenguaje C; cuando una constante de carácter como

```
"hola\n"
```

aparece en un programa en C, se almacena como un arreglo que contiene los caracteres de la cadena y termina con un `\0` para marcar el fin.

```
h o l a \n \0
```

La especificación de formato `%s` dentro de `printf` espera que el argumento correspondiente sea una cadena representada de este modo; `copy`' también se

basa en el hecho de que su argumento de entrada se termina con `\0`, y copia este carácter dentro del argumento de salida.

Todo esto implica que `\0` no es parte de un texto normal.

Es útil mencionar de paso que aun un programa tan pequeño como éste presenta algunos problemas de diseño. Por ejemplo, ¿qué debe hacer `main` si encuentra una línea que es mayor que su límite? `getline` trabaja en forma segura, en ese caso detiene la recopilación cuando el arreglo está lleno, aunque no encuentre el carácter nueva línea. Probando la longitud y el último carácter devuelto, `main` puede determinar si la línea fue demasiado larga, y entonces realiza el tratamiento que se desee. Por brevedad, hemos ignorado el asunto.

No existe forma para un usuario de `getline` de saber con anticipación cuán larga podrá ser una línea de entrada, por lo que `getline` revisa un posible desbordamiento ("overflow"). Por otro lado, el usuario de `copy` ya conoce (o lo puede averiguar) cuál es el tamaño de la cadena, por lo que decidimos no agregar comprobación de errores en ella.

- **Ejercicio 1-16**. Corrija la rutina principal del programa de la línea más larga de modo que imprima correctamente la longitud de líneas de entrada arbitrariamente largas, y tanto texto como sea posible. □
- **Ejercicio 1-17**. Escriba un programa que imprima todas las líneas de entrada que sean mayores de 80 caracteres. □
- **Ejercicio 1-18**. Escriba un programa que elimine los blancos y los tabuladores que estén al final de cada línea de en trad a, y que borre completamente las líneas en blanco. □
- **Ejercicio 1-19**. Escriba una función `reverse(s)` que invierta la cadena de caracteres `s`. Usela para escribir un programa que invierta su entrada, línea a línea. □

1.10 Variables externas y alcance

Las variables que están en `main`, tal como `line`, `longest`, etc., son privadas o locales a ella. Debido a que son declaradas dentro de `main`, ninguna otra función puede tener acceso directo a ellas. Lo mismo también es válido para variables de otras funciones; por ejemplo, la variable `i` en `getline` no tiene relación con la `i` que está en `copy`. Cada variable local de una función comienza a existir sólo cuando se llama a la función, y desaparece cuando la función termina. Esto es por lo que tales variables son conocidas como *variables automáticas*, siguiendo la terminología de otros lenguajes. Aquí se utilizará en adelante el término automático para hacer referencia a esas variables locales. (En el capítulo 4 se discute la categoría de almacenamiento `static` ("estática"), en la que las variables locales sí conservan sus valores entre llamadas.)

Puesto que las variables locales aparecen y desaparecen con la invocación de funciones, no retienen sus valores entre dos llamadas sucesivas, y deben ser inicializadas explícitamente en cada entrada. De no hacerlo, contendrán "basura" .

Como una alternativa a las variables automáticas, es posible definir variables que son *externas* a todas las funciones, esto es, variables a las que toda función puede tener acceso por su nombre. (Este mecanismo es parecido al `COMMON` de Fortran o a las variables de Pascal declaradas en el bloque más exterior). Debido a que es posible tener acceso global a las variables externas, éstas pueden ser usadas en lugar de listas de argumentos para comunicar datos entre funciones. Además, puesto que las variables externas se mantienen permanentemente en existencia, en lugar de aparecer y desaparecer cuando se llaman y terminan las funciones, mantienen sus valores aun después de que regresa la función que los fijó.

Una variable externa debe *definirse*, exactamente una vez, fuera de cualquier función; esto fija un espacio de almacenamiento para ella. La variable también debe *declararse* en cada función que desee tener acceso a ella; esto establece el tipo de la variable. La declaración debe ser una proposición `extern` explícita, o bien puede estar implícita en el contexto. Para concretar la discusión, reescribamos el programa de la línea más larga con `line`, `longest` y `max` como variables externas. Esto requiere cambiar las llamadas, declaraciones y cuerpos de las tres funciones.

```
#include <stdio.h>

#define MAXLINE 1000      /* máximo tamaño de una línea de entrada */

int max;                  /* máxima longitud vista hasta el momento */
char line [MAXLINE];      /* línea de entrada actual */
char longest [MAXLINE];   /* la línea más larga se guarda aquí */

int getline(void);
void copy(void);

/* imprime la línea de entrada más larga; versión especializada */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0) /* hubo una línea */
        printf("%s", longest);
    return 0;
}

/* getline: versión especializada */
```

```
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE - 1
        && (c=getchar()) != EOF && c != '\n'; ++i)
            line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
}
    line[i] = '\0';
    return i;
}

/* copy: versión especializada */
void copy(void)
{
    int i;
    extern char line[], longest[];

    i = 0;
    while ((longest[i] = line[i]) != '\0')
        ++i;
}
```

Las variables externas de `main`, `getline` y `copy` están definidas en las primeras líneas del ejemplo anterior, lo que establece su tipo y causa que se les asigne espacio de almacenamiento. Desde el punto de vista sintáctico, las definiciones externas son exactamente como las definiciones de variables locales, pero puesto que ocurren fuera de las funciones, las variables son externas. Antes de que una función pueda usar una variable externa, se debe hacer saber el nombre de la variable a la función. Una forma de hacer esto es escribir una declaración `extern` dentro de la función; la declaración es la misma que antes, excepto por la palabra reservada `extern`.

Bajo ciertas circunstancias, la declaración `extern` puede omitirse. Si la definición de una variable externa ocurre dentro del archivo fuente antes de su uso por una función en particular, entonces es necesario usar una declaración `extern` dentro de la función. La declaración `extern` en `main`, `getline` y `copy` es - por tanto - redundante. De hecho, una práctica común consiste en colocar las definiciones de todas las variables externas al principio del archivo fuente y luego omitir todas las declaraciones `extern`.

Si el programa está conformado por varios archivos de código fuente y una variable se define en `archivo1` y también se recurre a ella en `archivo2` y `archivo3`, entonces es necesario realizar las declaraciones `extern` en `archivo2` y `archivo3` para conectar las ocurrencias de la variable. La práctica común es reunir dichas declaraciones de variables y funciones `extern` en un archivo separado - históricamente denominado `header.h` (el sufijo `.h` se usa por convención para nombres de *header*) - los cuales son incluido por medio de `#include` al principio de cada archivo fuente. Las funciones de la biblioteca

estándar, por ejemplo, están declaradas en headers como `<stdio.h>`. Este tema se trata ampliamente en el capítulo 4, y la biblioteca en el capítulo 7 y en el apéndice B.

Puesto que las versiones especializadas de `getline` y `copy` no tienen argumentos, la lógica sugeriría que sus prototipos al principio del archivo deben ser `getline()` y `copy()`. Pero para mantener compatibilidad con programas de C anteriores, el estándar considera una lista vacía como una declaración del estilo antiguo, y suspende toda revisión de listas de argumentos; para una lista explícitamente vacía debe emplearse la palabra `void`. Esto se discutirá en el capítulo 4.

Se debe notar que en esta sección empleamos cuidadosamente las palabras *definición* y *declaración* cuando nos referimos a variables externas. La palabra "definición" se refiere al lugar donde se crea la variable o se le asigna un lugar de almacenamiento; "declaración" se refiere al lugar donde se establece la naturaleza de la variable pero no se le asigna espacio.

A propósito, existe una tendencia a convertir todo en variables `extern`, debido a que aparentemente simplifica las comunicaciones — las listas de argumentos son cortas y las variables siempre están allí, cuando se las necesita. Pero las variables externas existen siempre, aun cuando no hacen falta. Descansar en la dependencia de las variables externas resulta peligroso, puesto que lleva a programas cuyas conexiones entre datos no son absolutamente obvias — las variables pueden alterarse de manera inadvertida e inesperada, y dichos programas son difíciles de modificar. La segunda versión del programa de la línea mayor es inferior a la primera, en parte por las anteriores razones y en parte porque destruye la generalidad de dos útiles funciones, introduciendo en ellas los nombres de las variables que manipula.

Hasta este punto hemos descrito lo que podría llamarse los *fundamentos convencionales de C*. Con estos fundamentos, le será posible escribir programas útiles de tamaño considerable, y probablemente sería una buena idea hacer una pausa suficientemente grande para realizarlos. Estos ejercicios sugieren programas de complejidad algo mayor que los presentados anteriormente en este capítulo.

- **Ejercicio 1-20**. Escriba un programa `detab` que reemplace tabuladores de la entrada con el número apropiado de caracteres en blancos para espaciar hasta el siguiente paro de tabulación. Considere un conjunto fijo de paros de tabulación, digamos cada *n* columnas. ¿Debe ser *n* una variable o un parámetro simbólico? □
- **Ejercicio 1-21**. Escriba un programa `entab` que reemplace cadenas de blancos por el mínimo número de tabuladores y blancos para obtener el mismo espaciado. Considere los paros de tabulación de igual manera que para `detab`. Cuando un tabulador o un simple espacio en blanco fuese suficiente para alcanzar un paro de tabulación, ¿a cuál se le debe dar preferencia? □

- **Ejercicio 1-22**. Escriba un programa para “dividir” líneas grandes de entrada en dos o más líneas más cortas después del último carácter no blanco que ocurra antes de la n-ésima columna de entrada. Asegúrese de que su programa se comporte apropiadamente con líneas muy largas, y de que no existan caracteres en blancos o tabuladores antes de la columna especificada. □
- **Ejercicio 1-23**. Escriba un programa para eliminar todos los comentarios de un programa en C. No olvide manejar apropiadamente las cadenas entre comillas y las constantes de carácter. Los comentarios de C no se anidan. □
- **Ejercicio 1-24**. Escriba un programa para revisar los errores de sintaxis rudimentarios de un programa en C, como paréntesis, llaves y corchetes no alineados. No olvide las comillas ni los apóstrofos, las secuencias de escape y los comentarios. (Este programa es difícil si se hace completamente general). □

Capitulo 2: Tipos, Operadores y Expresiones

Las variables y las constantes son los objetos de datos básicos que se manipulan en un programa. Las declaraciones muestran las variables que se van a utilizar y establecen el tipo que tienen y algunas veces cuáles son sus valores iniciales. Los operadores especifican lo que se hará con las variables. Las expresiones combinan variables y constantes para producir nuevos valores. El tipo de un objeto determina el conjunto de valores que puede tener y qué operaciones se pueden realizar sobre él. Estos son los temas de este capítulo.

El estándar ANSI ha hecho muchos pequeños cambios y agregados a los tipos básicos y a las expresiones. Ahora hay formas `signed` y `unsigned` de todos los tipos enteros, y notaciones para constantes sin signo y constantes de carácter hexadecimales. Las operaciones de coma flotante pueden hacerse en precisión sencilla; también hay un tipo `long double` para precisión extendida. Las constantes de cadena pueden concatenarse al tiempo de comilación. Las enumeraciones son ya parte del lenguaje, formalizando una característica pendiente por mucho tiempo. Los objetos pueden ser declarados `const`, lo que impide que cambien. Las reglas para conversión automática entre tipos aritméticos fueron aumentadas para manejar el conjunto de tipos más rico actual.

2.1 Nombres de variables

Aunque no lo mencionamos en el capítulo 1, existen algunas restricciones en los nombre de las variables y de las constantes simbólicas. Los nombres se componen de letras y dígitos; el primer carácter debe ser una letra. El carácter de subrayado “_” cuenta como una letra; algunas veces es útil para mejorar la legibilidad de nombres largos de variables. Sin embargo, no se debe comenzar los nombres de variables con este carácter, puesto que las rutinas de biblioteca con frecuencia usan tales nombres. Las letras mayúsculas y minúsculas son distintas, de tal manera que `x` y `X` son dos nombres diferentes. La práctica tradicional de C es usar letras minúsculas para nombres de variables, y todo en mayúsculas para constantes simbólicas.

Al menos los primeros 31 caracteres de un nombre interno son significativos, para nombres de funciones y variables externas el número puede ser menor que 31, puesto que los nombres externos los pueden usar los ensambladores y los cargadores, sobre los que el lenguaje no tiene control. Para nombres externos, el estándar garantiza distinguir sólo 6 caracteres (y sin diferenciar mayúsculas de minúsculas). Las palabras clave como `if`; `else`, `int`, `float`, etc., se encuentran reservadas: no se pueden utilizar como nombres de variables. Todas ellas deben escribirse con minúsculas.

Es conveniente elegir nombres que estén relacionados con el propósito de la variable, que no sea probable confundirlos tipográficamente. Por estilo,

nosotros tendemos a utilizar nombres cortos para variables locales (especialmente índices de iteraciones), y nombres más largos para variables externas.

2.2 Tipos y tamaños de datos

Hay unos cuantos tipos de datos básicos en C:

`char`	un solo byte, capaz de contener un carácter del conjunto de caracteres local.
`int`	un entero, normalmente del tamaño natural de los enteros en la máquina en la que se ejecuta.
`float`	punto flotante de precisión normal.
`double`	punto flotante de doble precisión.

Además, existen algunos calificadores que se aplican a estos tipos básicos, `short` y `long` se aplican a enteros:

```
short int sh;
long int counter;
```

La palabra `int` puede omitirse de tales declaraciones, lo que típicamente se hace.

La intención es que `short` y `long` puedan proporcionar diferentes longitudes de enteros donde sea práctico; `int` será normalmente el tamaño natural para una máquina en particular. A menudo `short` es de 16 bits y `long` de 32; `int` es de 16 o de 32 bits. Cada compilador puede seleccionar libremente los tamaños apropiados para su propio hardware, sujeto sólo a la restricción de que los `short`s e `int`s son, por lo menos - de 16 bits, los `long`s son por lo menos de 32 bits y el `short` no es mayor que `int`, el cual a su vez no es mayor que `long`.

El calificador `signed` o `unsigned` puede aplicarse a `char` o a cualquier entero. Los números *unsigned* son siempre positivos o cero y obedecen las leyes de la aritmética módulo 2“, donde *n* es el número de bits en el tipo. Así, por ejemplo, si los `char` son de 8 bits, las variables `unsigned char` guardan valores entre `0` y `255`, en tanto que las variables `signed char` guardan valores entre `-128` y `127` (en una máquina de complemento a dos). El hecho de que los `char`s ordinarios sean con signo o sin él depende de la máquina, pero los caracteres que se pueden imprimir son siempre positivos.

El tipo `long double` especifica coma flotante de precisión extendida. Igual que con los enteros, los tamaños de objetos de coma flotante se definen en la implantación; `float`, `double` y `long double` pueden representar uno, dos o tres tamaños distintos.

Los archivos de encabezado headers estándar `<limits.h>` y `<float.h>` contienen constantes simbólicas para todos esos tamaños, junto con otras propiedades de la máquina y del compilador, los cuales se discuten en el apéndice B.

- **Ejercicio 2-1**. Escriba un programa para determinar los rangos de variables `char`, `short`, `int` y `long`, tanto `signed` como `unsigned`, imprimiendo los valores apropiados de los headers estándar y por cálculo directo. Es más difícil si los calcula: determine los rangos de los varios tipos de punto flotante. □

2.3 Constantes

Una constante entera como `1234` es un `int`. Una constante `long` se escribe con una `1` (ele) o `L` terminal, como en `123456789L`; un entero demasiado grande para caber dentro de un `int` también será tomado como `long`. Las constantes sin signo se escriben con una `u` o `U` final, y el sufijo `ul` o `UL` denota `unsigned long`.

Las constantes de punto flotante contienen un punto decimal (`123.4`) o un exponente (`1e-2`) o ambos; su tipo es `double`, a menos que tengan sufijo. Los sufijos `f` o `F` indican una constante `float`; `l` o `L` indican un `long double`.

El valor de un entero puede especificarse en forma octal o hexadecimal en lugar de decimal. Un `0` (cero) al principio de una constante entera significa octal; `0x` ó `0X` al principio significa hexadecimal. Por ejemplo, el decimal `31` puede escribirse como `037` en octal y `0x1f` ó `0x1F` en hexadecimal. Las constantes octales y hexadecimales también pueden ser seguidas por `L` para convertirlas en `long` y `U` para hacerlas `unsigned`: `OXFUL` es una constante `unsigned long` con valor de `15` en decimal.

Una *constante de carácter* es un entero, escrito como un carácter dentro de apóstrofos, tal como `'x'`. El valor de una constante de carácter es el valor numérico del carácter en el conjunto de caracteres de la máquina. Por ejemplo, en el conjunto de caracteres el carácter constante `'0'` tiene el valor de `48`, el cual no está relacionado con el valor numérico `0`. Si escribimos `'0'` en vez de un valor numérico tal como 48 (que depende del conjunto de caracteres), el programa resulta independiente del valor particular y será más fácil de leer. Las constantes de carácter participan en operaciones numéricas tal como cualesquier otros enteros, aunque se utilizan más comúnmente en comparaciones con otros caracteres.

Ciertos caracteres pueden ser representados en constante de carácter y de cadena, Por medio de secuencias de escape como `\n` (caraceter *nueva línea*); esas secuencias se ven como dos caracteres, pero representan sólo uno. Además, un patrón de bits arbitrario de tamaño de un byte puede ser especificado por

```
'\ooo'
```

en donde `ooo` son de uno a tres dígitos octales (`0`…`7`) o por

```
'\xhh'
```

en donde hh son uno o más dígitos hexadecimales (0…9, a…f, A…F). Así podríamos escribir

```
#define VTAB '\013'   /* tab vertical ASCII */
#define BELL '\007'   /* carácter campana ASCII */
```

o, en hexadecimal,

```
#define VTAB '\xb'    /* tab vertical ASCII */
#define BELL '\x7'    /* carácter campana ASCII */
```

La constante de carácter '\0' representa el **carácter nulo** (con valor cero). '\0' a menudo se escribe en vez de 0 para enfatizar la naturaleza de carácter de algunas expresiones, pero el valor numérico es precisamente 0.

El conjunto completo de secuencias de escape es

Secuencia de Escape	Caracter
\a	carácter de alarma/campana (**BELL**)
\b	retroceso, **BKSP**
\f	avance de hoja, **FF**
\n	nueva línea, **LF**
\r	regreso de carro, **CR**
\t	tabulador horizontal, **TAB**
\v	tabulador vertical, **VTAB**
\0	caracter nulo, **NULL**
\\	barra invertida \
\?	signo de interrogación **?**
\'	apóstrofo '
\"	comillas “
\ooo	Número octal
\xhh	Número hexadecimal

Una *expresión constante* es una expresión que sólo inmiscuye constantes. Tales expresiones pueden ser evaluadas durante la compilación en vez de que se haga en tiempo de ejecución, y por tanto pueden ser utilizadas en cualquier lugar en que pueda encontrarse una constante, como en

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

o

```
#define BISIESTO 1 /* en años bisiestos */
int days[31+28+BISIESTO+31+30+31+30+31+31+30+31+30+31];
```

Una *constante de cadena* o *cadena literal*, es una secuencia de cero o más caracteres encerrados entre comillas, como en

```
"Soy una cadena"
```

o

```
""/* la cadena vacía */
```

Las comillas no son parte de la cadena, sólo sirven para delimitarla. Las mismas secuencias de escape utilizadas en constantes de carácter se aplican en cadenas; `\"` representa el carácter comillas. Las constantes de cadena pueden ser concatenadas en tiempo de compilación:

```
"Viva " "Peron!"
```

es equivalente a

```
"Viva Peron!"
```

Esto es útil para separar cadenas largas entre varias líneas de código fuente.

Técnicamente, una constante de cadena es un arreglo de caracteres. La representación interna de una cadena tiene un carácter nulo `'\0'` al final, de modo que el almacenamiento físico requerido es uno más del número de caracteres escritos entre las comillas. Esta representación significa que no hay límite en cuanto a qué tan larga puede ser una cadena, pero los programas deben leer completamente una cadena para determinar su longitud. La función `strlen(s)` de la biblioteca estándar regresa la longitud de su argumento `s` de tipo cadena de caracteres, excluyendo el `'\0'` del final. Aquí está nuestra versión:

```
/* strlen:    /* regresa la longitud de s */
int strlen(char s[])
{
    int i;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

`strlen` y otras funciones para cadenas están declaradas en el *header* estándar `<string.h>`.

Se debe ser cuidadoso al distinguir entre una constante de carácter y una cadena que contiene un sólo carácter: `'x'` no es lo mismo que `"x"`. El primero es un entero, utilizado para producir el valor numérico de la letra x en el conjunto de caracteres de la máquina. El último es un arreglo de caracteres que contiene un carácter (el caracter `x`) y un caracter `'\0'` al final.

Existe otra clase de constante, la *constante de enumeración*. Una enumeración es una lista de valores enteros constantes, como en

```
enum boolean {NO, SI};
```

El primer nombre en un `enum` tiene valor `0`, el siguiente `1`, y así sucesivamente, a menos que sean especificados valores explícitos. Si no son especificados

todos los valores, los valores no especificados continúan la progresión a partir del último valor que si lo fue, como en el segundo de esos ejemplos:

```
enum escapes { BELL = '\a', RETROCESO = '\b', TAB = '\t',
               NVALIN = '\n', VTAB = '\v' , RETURN = '\r'};

enum months { ENE = 1, FEB, MAR, ABR, MAY, JUN,
              JUL, AGO, SEP, OCT, NOV, DIC};
            /* FEB es 2, MAR es 3, etc. */
```

Los nombres que están en enumeraciones diferentes deben ser distintos. Los valores no necesitan ser distintos d entro de la misma enumeración.

Las enumeraciones proporcionan una manera conveniente de asociar valores constantes con nombres, una alternativa a `#define` con la ventaja de que los valores pueden ser generados por uno mismo. Aunque las variables de tipos `enum` pueden declararse, los compiladores no necesitan revisar que lo que se va a almacenar en tal variable es un valor válido para la enumeración. No obstante, las variables de enumeración ofrecen la oportunidad de revisarlas (y a menudo tal cosa es mejor que los `#define`). Además, un depurador puede ser capaz de imprimir los valores de variables de enumeración en su forma simbólica.

2.4 Declaraciones

Todas las variables deben ser declaradas antes de su uso, aunque ciertas declaraciones pueden ser hechas en forma implícita por el contexto. Una declaración especifica un tipo, y contiene una lista de una o más variables de ese tipo, como en

```
int inferior, superior, paso;
char c, line [1000];
```

Las variables pueden ser distribuidas entre las declaraciones en cualquier forma; la lista de arriba podría igualmente ser escrita como

```
int  inferior;
int  superior;
int  paso;
char c;
char line [1000];
```

Esta última forma ocupa más espacio, pero resulta conveniente para agregar un comentario a cada declaración o para realizar subsecuentes modificaciones.

Una variable también puede ser inicializada en su declaración. Si el nombre es seguido por un signo de igual y una expresión, la expresión sirve como un inicializador, como en

```
char  esc = '\\';
int   i = 0;
int   limit = MAXLINE + 1;
float eps = l.0e—5;
```

Si la variable en cuestión no es automática, la inicialización es efectuada sólo una vez, conceptualmente antes de que el programa inicie su ejecución, y el inicializador debe ser una expresión constante. Una variable automática explícitamente inicializada es inicializada cada vez que se entra a la función o bloque en que se encuentra; el inicializador puede ser cualquier expresión. Las variables estáticas y externas son inicializadas en cero por omisión. Las variables automáticas para las que no hay un inicializador explícito tienen valores indefinidos (esto es, basura).

El calificador `const` puede aplicarse a la declaración de cualquier variable para especificar que su valor no será cambiado. Para un arreglo, el calificador `const` indica que los elementos no serán alterados.

```
const double e = 2.71828182845905;
const char msg[] = "precaución: ";
```

La declaración `const` también se puede utilizar con argumentos de tipo arreglo, para indicar que la función no cambia ese arreglo:

```
int strlen(const char[]);
```

Si se efectúa un intento de cambiar un `const`, el resultado está definido por la implantación.

2.5 Operadores aritméticos

Los operadores aritméticos binarios son `+`, `-`, `*`, y `/`, y el operador módulo `%`. La división entera trunca cualquier parte fraccionaria. La expresión

```
x % y
```

produce el residuo cuando `x` es dividido entre `y`, por lo que es cero cuando `y` divide a `x` exactamente. Por ejemplo, un año es bisiesto si es divisible entre 4 pero no entre 100, excepto aquellos años que si son divisibles entre 400, que *si son* bisiestos. Por lo tanto

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d es un año bisiesto\n", year);
else
    printf("%d no es un año bisiesto\n", year);
```

El operador `%` no puede aplicarse a operandos `float` o `double`. La dirección de truncamiento para `/` y el signo del resultado de `%` son dependientes de la máquina para operandos negativos, así como la acción que se toma en caso de sobreflujo o subflujo.

Los operadores binarios `+` y `-` tienen la misma precedencia, la cual es menor que la precedencia de `*`, `/`, y `%`, que a su vez es menor que `+` y `-` unarios. Los operadores aritméticos se asocian de zquierda a derecha.

La tabla 2-1 que se encuentra al final de este capítulo, resume la precedencia y asociatividad para todos los operadores.

2.6 Operadores de relación y lógicos

Los *operadores de relación* son

>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que

Todos ellos tienen la misma precedencia. Precisamente bajo ellos en precedencia están los *operadores de igualdad*:

==	Exactamente igual a
=!	Exactamente no igual a

Los operadores de relación tienen precedencia inferior que los operadores aritméticos, así que una expresión como `i < lim -1` se toma como `i < (lim -1 )`, como se esperaría.

Más interesantes son los *operadores lógicos* && y `||`. Las expresiones conectadas por && o `||` son evaluadas de izquierda a derecha, y la evaluación se detiene tan pronto como se conoce el resultado verdadero o falso. La mayoría de los programas en C descansan sobre esas propiedades. Por ejemplo, aquí está un ciclo de la función de entrada `getline` que escribimos en el capítulo 1:

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Antes de leer un nuevo carácter es necesario verificar que hay espacio para almacenarlo en el arreglo `s`, así que la prueba `i < lim -1` *debe* hacerse primero. Además, si esta prueba falla, no debemos seguir y leer otro carácter.

De manera semejante, seria desafortunado si `c` fuese probada contra antes de que se llame a `getchar`; por lo tanto, la llamada y la asignación deben ocurrir antes de que se pruebe el carácter `c`.

La precedencia de && es más alta que la de `||`, y ambas son menores que los operadores de relación y de asignación, así que expresiones como

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

no requieren de paréntesis adicionales. Pero puesto que la precedencia de `!=` es superior que la asignación, los paréntesis se necesitan en

```
(c=getchar()) != '\n'
```

para obtener el resultado deseado de asignación a `c` y después comparación con `'\n'`.

Por definición, el valor numérico de una expresión de relación o lógica es `1` si la relación es verdadera, y `0` si la relación es falsa.

El operador unario de negación `!` convierte a un operando que no es cero en `0`, y a un operando cero en `1`. Un uso común de `!` es en construcciones como

```
if (!nvalido)
```

en lugar de

```
if (valido == 0)
```

Es difícil generalizar acerca de cuál es la mejor. Construcciones como `!nvalido` se leen en forma agradable (“si es inválido”), pero otras más complicadas pueden ser difíciles de entender.

- **Ejercicio 2-2**. Escriba un ciclo equivalente a la iteración `for` anterior sin usar && o `!!`.

2.7 Conversiones de tipo

Cuando un operador tiene operandos de tipos diferentes, éstos se convierten a un tipo común de acuerdo con un reducido núm ero de reglas. En general, las únicas conversiones automáticas son aquellas que convierten un operando “angosto” en uno “amplio” sin pérdida de información, tal como convertir un entero a coma flotante en una expresión como `f + i`. Las expresiones que no tienen sentido, como utilizar un `float` como subíndice, no son permitidas. Las expresiones que podrían perder información, como asignar un tipo mayor a uno más corto, o un tipo de coma flotante a un entero, pueden producir una advertencia, pero no son ilegales.

Un `char` sólo es un entero pequeño, por lo que los `char`s se pueden utilizar libremente en expresiones aritméticas. Esto permite una flexibilidad considerable en ciertas clases de transformación de caracteres. Una es ejemplificada con esta ingenua implantación de la función `atoi`, que convierte una cadena de dígitos en su equivalente numérico.

```
/* atoi: convierte s en entero */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Tal com o se discutió en el capítulo 1, la expresión

```
s[i] - '0'
```

da el valor numérico del carácter almacenado en `s[i]`, debido a que los valores de '0', 1, etc., forman una secuencia ascendente contigua.

Otro ejemplo de conversión de `char` a `int` es la función `lower`, que conviene un carácter simple a minúscula en el conjunto de caracteres . Si el carácter no es una letra mayúscula, `lower` lo regresa sin cambio.

```
/* lower: convierte c a minúscula; solamente ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Esto funciona en puesto que las correspondientes letras mayúsculas y minúsculas se encuentran a una distancia fija como valores numéricos y cada alfabeto es contiguo (no hay sino letras entre `A` y `Z`). Sin embargo, esta última observación no es cierta para el conjunto de caracteres EBCDIC, así que este código podría convertir algo más que sólo letras en EBCDIC.

El header estándar `<ctype.h>`, que se describe en el apéndice B, define una familia de funciones que proporcionan pruebas y conversiones independientes de los juegos de caracteres. Por ejemplo, si `c` es una mayúscula, la función `tolower(c)` regresa el valor de la letra minúscula de `c` (de modo que `tolower` es un reemplazo transportable para la función `lower` mostrada antes). De modo semejante, la prueba

```
c >= '0' && c <= '9'
```

puede reemplazarse por

```
isdigit(c)
```

Nosotros utilizaremos las funciones de `<ctype.h>` en adelante. Existe un sutil punto acerca de la conversión de caracteres a enteros.' El lenguaje no especifica si las variables de tipo `char` son valores con o sin signo. Cuando un `char` se convierte a `int`, ¿puede producir alguna vez un entero negativo?

La respuesta varía de una máquina a otra, reflejando diferencias en la arquitectura. En algunas máquinas un `char` cuyo bit más a la izquierda es `1` se convertirá a un entero negativo (“extensión de signo”). En otras, un `char` resulta promovido a un `int` agregando ceros del lado izquierdo, así que siempre es positivo.

La definición de C garantiza que ningún carácter que esté en el conjunto estándar de caracteres de impresión de la máquina será negativo, de modo que esos caracteres siempre serán cantidades positivas en las expresiones. Pero hay patrones arbitrarios de bits almacenados en variables de tipo carácter que pueden aparecer como negativos en algunas máquinas, aunque sean positivos en otras - por razones de portabilidad - se debe especificar `signed` o `unsigned` si se van a almacenar datos que no son caracteres en variables tipo `char`.

Las expresiones de relación como `i > j` y las expresiones lógicas conectadas por && y || están definidas para tener un valor de `1` siendo verdaderas, y `0` al ser falsas. De este modo, la asignación

```
d = c >= '0' && c <= '9'
```

vuelve `1` a `d` si `c` es un dígito, y `0` si no lo es. Sin embargo, las funciones como `isdigit` pueden regresar cualquier valor diferente de cero como verdadero. En el componente de validación de `if`, `while`, `for`, etc., "verdadero" sólo significa "diferente de cero", por lo que esto no hace diferencia.

Las conversiones aritméticas implícitas trabajan como se espera. En general, si un operador como `-f` o * que toma dos operandos (operador binario) tiene operandos de diferentes tipos, el tipo "menor" es *promovido* al tipo "superior" antes de que la operación proceda. El resultado es el del tipo mayor. La sección 6 del apéndice A establece las reglas de conversión en forma precisa. Si no hay operandos `unsigned`, sin embargo, el siguiente conjunto informal de reglas bastará:

- Siendo cualquier operando `long double`, conviértase el otro a `long double`.
- De otra manera, siendo cualquier operando `double`, conviértase el otro a `double`.
- De otra manera, siendo cualquier operando `float`, conviértase el otro a `float`.
- De otra manera, conviértase `char` y `short` a `int`.
- Luego, siendo cualquier operando `long`, conviértase el otro a `long`.

Nótese que los `float`s que están en una expresión no se convierten automáticamente a `double`; esto es a resultas de una alteración a la definición original. En general, las funciones matemáticas como las de `<math.h>` utilizarán doble precisión. La razón principal para usar `float` es ahorrar espacio de almacenamiento en arreglos grandes o - con menor frecuencia - ahorrar tiempo en procesamiento en máquinas donde la aritmética de doble precisión resulta particularmente costosa.

Cuando hay operandos `unsigned` las reglas de conversión son más complicadas. El problema es que las comparaciones de valores con signo y sin signo son dependientes de la máquina, debido a su dependencia de los tamaños de los varios tipos de enteros. Por ejemplo, supóngase que `int` es de 16 bits y `long` de 32. Entonces `-1L < 1U`, puesto que `1U`, que es un `unsigned int`, es promovido a `signed long`. Pero `-1L > 1UL`, puesto que `−1L` es promovido a `unsigned long`. Y así parece ser un gran número positivo.

Las conversiones también tienen lugar en las asignaciones; el valor del lado derecho es convertido al tipo de la izquierda, el cual es el tipo del resultado.

Un carácter es convertido a un entero, tenga o no extensión de signo, como se describió anteriormente.

Los enteros más largos son convertidos a cortos o a `char` desechando el exceso de bits de más alto orden. Así en

```
int  i;
char c;

i = c;
c = i;
```

el valor de `c` no cambia. Esto es verdadero ya sea que se inmiscuya o no la extensión de signo. Sin embargo, el invertir el orden de las asignaciones podría producir pérdida de información.

Si `x` es `float` e `i` es `int`, entonces `x = i` e `i = x` producirán conversiones; de `float` a `int` provoca el truncamiento de cualquier parte fraccionaria. Cuando `double` se convierte a `float`, el que se redondee o trunque el valor es dependiente de la implantación.

Puesto que un argumento de la llamada a una función es una expresión, también suceden conversiones de tipo cuando se pasan argumentos a funciones. En ausencia del prototipo de una función, `char` y `short` pasan a ser `int`, y `float` se hace `double`. Esta es la razón por la que se han declarado los argumentos a funciones como `int` y `double`, aun cuando la función se llama con `char` y `float`.

Finalmente, la conversión explícita de tipo puede ser forzada (“coaccionada”) en cualquier expresión, con un operador unario llamado *cast*. En la construcción

```
(nombre-de-tipo) expresión
```

la *expresión* es convertida al tipo nombrado, según las reglas de conversión anteriores. El significado preciso de un *cast* es como si la *expresión* fuera asignada a una variable del tipo especificado, que se utiliza entonces en lugar de la construcción completa. Por ejemplo, la rutina de biblioteca `sqrt` espera un argumento de doble precisión `double`, y si maneja inadvertidamente algo diferente producirá resultados sin sentido (`sqrt` está declarado en `<math.h>`). Así, si `n` es un entero, podemos usar

```
sqrt((double) n)
```

para convertir el valor de `n` a doble antes de pasarlo a `sqrt`. Nótese que la conversión forzosa produce el *valor* de `n` en el tipo apropiado; `n` en sí no se altera. El operador *cast* tiene la misma alta precedencia que otros operadores unarios, como se resume en la tabla del final de este capítulo.

Si un prototipo de función declara argumentos, como debe ser normalmente, la declaración produce conversión forzada automática de los argumentos cuando la función es llamada. Así, dado el prototipo de la función `sqrt`:

```
double sqrt(double)
```

la llamada

```
root2 = sqrt(2)
```

obliga al entero 2 a ser el valor `double`, esto es `2.0`, sin necesidad de ningún *cast*.

La biblioteca estándar incluye una implantación transportable de un generador de números pseudoalealorios, y una función para inicializar la semilla; lo primero ilustra un *cast*:

```
unsigned long int next = 1;

/* rand: regresa un entero pseudoaleatorio en 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

- **Ejercicio 2-3**. Escriba la función `htoi(s)`, que convierte una cadena de dígitos hexadecimales (incluyendo `0x` o `0X` en forma optativa) en su valor entero equivalente. Los dígitos permitidos son del `0` al `9`, de la `a` a la `f`, y de la `A` a la `F`. □

2.8 Operadores de incremento y decremento

El lenguaje C proporciona dos operadores poco comunes para incrementar y decrementar variables. El *operador de aumento* ++ agrega 1 a su operando, en tanto que el *operador de disminución* – le resta 1. Hemos usado frecuentemente ++ para incrementar variables, como en

```
if (c == '\n')
   ++nl;
```

El aspecto poco común es que ++ y – pueden ser utilizado como prefijos (antes de la variable, como en `+ + n )`, o como postfijos (después de la variable: (11 ++). En ambos casos, el efecto es incrementar `n`. Pero la expresión `++ n` incrementa a `n` antes de que su valor se utilice, en tanto que `n++` incrementa a `n` después de que su valor se ha empleado. Esto significa que en un contexto donde el valor está siendo utilizado, y no sólo el efecto, `++n` y `n++` son diferentes.

Si `n` es 5, entonces

```
x = n++;
```

asigna 5 a `x`, pero

```
x = ++n;
```

hace que x sea 6. En ambos casos, n se hace 6. Los operadores de incremento y decremento sólo pueden aplicarse a variables; una expresión como (i + j)+ + es ilegal.

Dentro de un contexto en donde no se desea ningún valor, sino sólo el efecto de incremento, como en

```
if (c == '\n')
    nl++;
```

prefijos y postfijos son iguales. Pero existen situaciones en donde se requiere específicam ente unou otro. Por ejemplo, considérese la función `squeeze(s,c)`, que elimina todas las ocurrencias del carácter `c` de una cadena `s`.

```
/* squeeze: borra todas las c de s */
void squeeze(char s[], int c)
{
    int i, j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
}
```

Cada vez que se encuentra un valor diferente de `c`, éste se copia en la posición actual `j`, y sólo entonces `j` es incrementada para prepararla para el siguiente carácter. Esto es exactamente equivalente a

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Otro ejemplo de construcción semejante viene de la función `getline` que escribimos en el capítulo 1, en donde podemos reemplazar

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

por algo más compacto como

```
if (c == '\n')
    s[i++] = c;
```

Como un tercer ejemplo, considérese que la función estándar `strcat(s,t)`, que concatena la cadena `t` al final de la cadena `s`. `strcat` supone que hay suficiente espacio en `s` para almacenar la combinación. Como la habíamos escrito, `strcat` no regresaba un valor; en cambio la versión de la biblioteca estándar regresa un apuntador a la cadena resultante.

```
/* strcat: concatena t al final de s; s debe ser suficientemente grande *1
void strcat(char s[], char t[])
```

```
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* encontrar fin de s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copiar t */
        ;
}
```

Como cada carácter es copiado de `t` a `s`, el ++ postfijo se aplica tanto a `i` como a `j` para estar seguros de que ambos están en posición para la siguiente iteración.

- **Ejercicio 2-4**. Escriba una versión alterna de `squeeze(sl,s2)` que borre cada carácter de `s1` que coincida con cualquier carácter de la *cadena* `s2`. □
- **Ejercicio 2-5**. Escriba la función `any(sl,s2)`, que regresa la primera posición de la cadena `s1` en donde se encuentre cualquier carácter de la cadena `s2`, o `-1` si `s1` no contiene caracteres de `s2`. (La función de biblioteca estándar `strpbrk` hace el mismo trabajo pero regresa un apuntador a la posición encontrada.) □

2.9 Operadores para manejo de bits

El lenguaje C proporciona seis operadores para manejo de bits; sólo pueden ser aplicados a operandos integrales, esto es, `char`, `short`, `int`, y `long`, con o sin signo.

&	*AND* de bits
|	*OR* inclusivo de bits
^	*OR* exclusivo de bits
<<	corrimiento a la izquierda
>>	corrimiento a la derecha
~	complemento a uno (unario)

El operador *AND* de bits & a menudo es usado para enmascarar algún conjunto de bits; por ejemplo,

```
n = n & 0177;
```

hace cero todos los bits de `n`, menos los 7 de menor orden.

El operador *OR* de bits | es empleado para encender bits:

```
x = x | SET_ON;
```

fija en uno a todos los bits de `x` que son uno en `SET_ON`.

El operador *OR* exclusivo ^ pone un uno en cada posición en donde sus operandos tienen bits diferentes, y cero en donde son iguales.

Se deben distinguir los operadores de bits & y | de los operadores lógicos && y ||, que implican evaluación de izquierda a derecha de un valor de verdad. Por ejemplo, si x es 1 y y es 2, entonces x & y es cero en tanto que x && y es uno.

Los operadores de corrimiento << y >> realizan corrimientos a la izquierda y a la derecha de su operando que está a la izquierda, el número de posiciones de bits dado por el operando de la derecha, el cual debe ser positivo. Así x << 2 desplaza el valor de x a la izquierda dos posiciones, llenando los bits vacantes con cero; esto es equivalente a una multiplicación por 4. El correr a la derecha una cantidad unsigned siempre llena los bits vacantes con cero. El correr a la derecha una cantidad signada llenará con bits de signo (“corrimiento aritmético”) en algunas máquinas y con bits 0 (“corrimiento lógico”) en otras.

El operador unario ~ da el complemento a uno de un entero; esto es, convierte cada bit 1 en un bit 0 y viceversa. Por ejemplo,

```
x = x & ~077
```

fija los últimos seis bits de x en cero. Nótese que x & ~077 es independiente de la longitud de la palabra, y por lo tanto, es preferible a, por ejemplo, x & 0177700, que supone que x es una cantidad de 16 bits. La forma transportable no involucra un costo extra, puesto que ~077 es una expresión constante que puede ser evaluada en tiempo de compilación.

Como ilustración de algunos de los operadores de bits, considere la función getbits(x,p,n) que regresa el campo de n bits de x (ajustado a la derecha) que principia en la posición p. Se supone que la posición del bit 0 está en el borde derecho y que n y p son valores positivos adecuados. Por ejemplo, getbits(x,4,3) regresa los tres bits que están en la posición 4, 3 y 2, ajustados a la derecha.

```
/* getbits: obtiene n bits desde la posición p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

La expresión x >> (p+1-n) mueve el campo deseado al borde derecho de la palabra. ~0 es todos los bits en 1; corriendo *n* bits hacia la izquierda con ~0<<n coloca ceros en los *n* bits más a la derecha; complementado con ~ hace una máscara de unos en los *n* bits más a la derecha.

- **Ejercicio 2-6**. Escriba una función setbits(x,p,n,y) que regresa x con los *n* bits que principian en la posición p iguales a los *n* bits más a la derecha de y, dejando los otros bits sin cambio. □
- **Ejercicio 2-7**. Escriba una función invert(x,p,n) que regresa x con los *n* bits que principian en la posición p invertidos (esto es, 1 cambiado a 0 y viceversa), dejando los otros sin cambio. □
- **Ejercicio 2-8**. Escriba una función rightrot(x,n) que regresa el valor del entero x rotado a la derecha *n* posiciones de bits. □

2.10 Operadores de asignación y expresiones

Las expresiones tales como

```
i = i + 2
```

en las que la variable del lado izquierdo se repite inmediatamente en el derecho, pueden ser escritas en la forma compacta

```
i += 2
```

El operador += se llama *operador de asignación*.

La mayoría de los operadores binarios (operadores como + que tienen un operando izquierdo y otro derecho) tienen un correspondiente operador de asignación *op=* , en donde *op* es uno de

```
+ - * / % << >> & * |
```

Si *expr1* y *expr2* son expresiones, entonces

```
expr1 op= expr2
```

es equivalente a

```
expr1 = (expr1) op (expr2)
```

exceptuando que *expr1* se calcula sólo una vez. Nótense los paréntesis alrededor de *expr2*:

```
x *= y + 1
```

significa

```
x = x * (y + 1)
```

Y no

```
x = x * y + 1
```

Como ejemplo, la función `bitcount` cuenta el número de bits en 1 en su argumento entero.

```
/* bitcount: cuenta bits 1 en x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Declarar al argumento `x` como `unsigned` asegura que cuando se corre a la derecha, los bits vacantes se llenarán con ceros, no con bits de signo, sin importar la máquina en la que se ejecute el programa.

Muy aparte de su concisión, los operadores de asignación tienen la ventaja de que corresponden mejor con la forma en que la gente piensa. Decimos “suma 2 a i” o “incrementa i en 2” , no “toma i, agrégale 2, después pon el resultado de nuevo en i”. Así la expresión `i += 2` es preferible a `i = i + 2`. Además, para una expresión complicada como

```
yyval[yypv[p3+p4] + yypv[p1]] += 2
```

el operador de asignación hace al código más fácil de entender, puesto que el lector no tiene que verificar arduamente que dos expresiones muy largas son en realidad iguales, o preguntarse por qué no lo son, y un operador de asignación puede incluso ayudar al compilador a producir código más eficiente.

Ya hemos visto que la proposición de asignación tiene un valor y puede estar dentro de expresiones; el ejemplo más común es

```
while ((c = getchar()) != EOF)
    ...
```

Los otros operadores de asignación (+=, −=, etc.) también pueden estar dentro de expresiones, aunque esto es menos frecuente.

En todas esas expresiones, el tipo de una expresión de asignación es el tipo de su operando del lado izquierdo, y su valor es el valor después de la asignación.

- **Ejercicio 2-9**. En un sistema de números de complemento a dos, `x &= (x-1)` borra el bit 1 de más a la derecha en `x`. Explique el porqué. Utilice esta observación para escribir una versión más rápida de `bitcount`. □

2.11 Expresiones condicionales

Las proposiciones

```
if (a > b)
    z = a;
else
    z = b;
```

calculan en `z` el máximo de `a` y `b`. La *expresión condicional*, escrita con el operador ternario ”`?:`“ proporciona una forma alternativa para escribir ésta y otras construcciones semejantes. En la expresión

```
expr1 ? expr2 : expr3
```

la expresión *expr1* es evaluada primero. Si es diferente de cero (verdadero), entonces la expresión *expr2* es evaluada, y ése es el valor de la expresión condicional. De otra forma, se evalúa *expr3*, y ése es el valor. Sólo se evalúa una de entre *expr2* y *expr3*. Asi, para hacer `z` el máximo de `a` y `b`,

```
z = (a > b) ? a : b;    /* z = max(a, b) */
```

Se debe notar que la expresión condicional es en sí una expresión, y se puede utilizar en cualquier lugar donde otra expresión pueda. Si *expr2* y *expr3* son de tipos diferentes, el tipo del resultado se determina por las reglas de conversión discutidas anteriormente en este capítulo. Por ejemplo, si `f` es un `float` y `n` es un `int`, entonces la expresión

```
(n > 0) ? f : n
```

es de tipo `float` sea `n` positivo o no.

Los paréntesis no son necesarios alrededor de la primera expresión de una expresión condicional, puesto que la precedencia de `?:` es muy baja, sólo arriba de la asignación. De cualquier modo son recomendables, puesto que hacen más fácil de ver la parte de condición de la expresión.

La expresión condicional frecuentemente lleva a un código conciso. Por ejemplo, este ciclo imprimen elementos de un arreglo, 10 por línea, con cada columna separada por un *caracter en blanco*, y con cada línea (incluida la última) terminada por un caracter *nueva línea*.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

Se imprime un carácter nueva línea después de cada diez elementos, y después del n-ésimo. Todos los otros elementos son seguidos por un *espacio en blanco*. Esto podría parecer complicado, pero es más compacto que el `if-else` equivalente. Otro buen ejemplo es

```
printf("Tiene %d elementos%s.\n", n, n==1 ? "" : "s");
```

- **Ejercicio 2-10**. Reescriba la función `lower`, que convierte letras mayúsculas e minúsculas, con una expresión condicional en vez de un `if-else`. □

2.12 Precedencia y orden de evaluación

La *tabla 2-1* resume las reglas de precedencia y asociatividad de todos los operadores, incluyendo aquellos que aún no se han tratado. Los operadores que están en la misma línea tienen la misma precedencia; los renglones están en orden de precedencia decreciente, así, por ejemplo, `%`, `/`, y * tienen todos la misma precedencia, la cual es más alta que la de + y - binarios. El “operador” `()` se refiere a la llamada a una función. Los operadores `->` y `.` son utilizados para tener acceso a miembros de estructuras; serán cubiertos en el capitulo 6, junto con `sizeof` (tamaño de un objeto). En el capitulo 5 se discuten * (indirección a través de un apuntador) y & (dirección de un objeto), y en el capítulo 3 se trata al operador `,` (coma).

Los +, -, y * unarios, tienen mayor precedencia que las formas binarias.

Nótese que la precedencia de los operadores de bits &, ^, y | están por debajo de == y != . Esto implica que las expresiones de prueba de bits como

```
if ((x & MASK) == 0) ...
```

deben ser completamente colocadas entre paréntesis para dar los resultados apropiados.

Como muchos lenguajes, C no especifica el orden en el cual los operandos de un operador serán evaluados. (Las excepciones son &&, ||, ?: y ,.) Por ejemplo, en proposiciones como

```
x = f() + g();
```

f puede ser evaluada antes de g o viceversa; de este modo si f o g alteran una variable de la que la otra depende, x puede depender del orden de evaluación. Se pueden almacenar resultados intermedios en variables temporales para asegurar una secuencia particular.

De manera semejante, el orden en el que se evalúan los argumentos de una función no está especificado, de modo que la proposición

```
printf("%d %d\n", ++n, power(2, n));    /* EQUIVOCADO */
```

puede producir resultados diferentes con distintos compiladores, dependiendo de si n es incrementada antes de que se llame a power. La solución, por supuesto, es escribir

```
++n;
printf("%d %d\n", n, power(2, n));
```

Las llamadas a funciones, proposiciones de asignación anidadas, y los operadores de incremento y decremento provocan “efectos colaterales” — alguna variable resulta modificada como producto de la evaluación de una expresión. En cualquier expresión que involucra efectos colaterales, pueden existir sutiles dependencias del orden en que las variables involucradas en la expresión se actualizan. La infortunada situación es tipificada por la proposición

```
a[i] = i++;
```

La pregunta es si el subíndice es el viejo o el nuevo valor de i. Los compiladores pueden interpretar esto en formas diferentes, y generar diferentes respuestas dependiendo de su interpretación. El estándar deja intencionalmente sin especificación la mayoría de estos aspectos. Cuando existen efectos colaterales (asignación a variables) dentro de una expresión, se deja a la prudencia del compilador, puesto que el orden más eficiente depende mayormente de la arquitectura de la máquina. (El estándar sí especifica que todos los efectos colaterales sobre argumentos sucedan antes de que la función sea llamada, pero eso podría no ayudar en la llamada a printf mostrada anteriormente).

La moraleja es que escribir un código dependiente del orden de evaluación es una mala práctica de programación en cualquier lenguaje. Naturalmente, es

necesario conocer qué cosas evitar, pero si no sabe cómo varias máquinas resuelven las cosas, no debe intentar sacar provecho de una implantación particular.

Tabla 2-1: Precedencia y asociatividad de operadores

Operadores	Asociatividad
`() [] -> .`	izquierda a derecha
`! ~ ++ – + - * (tipo) sizeof`	derecha a izquierda
`* / %`	izquierda a derecha
`+ -`	izquierda a derecha
`<< >>`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`&`	izquierda a derecha
`^`	izquierda a derecha
`|`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`?:`	derecha a izquierda
`= += -= *= /= %= &= ^= |= <<= >>=`	derecha a izquierda

Capítulo 3: Control de Flujo

Las proposiciones de control de flujo de un lenguaje especifican el orden en que se realiza el procesamiento. Ya hemos visto la mayoría de las construcciones de control de flujo en ejemplos anteriores; aquí completaremos el conjunto, y seremos más precisos acerca de las discutidas con anterioridad.

3.1 Proposiciones y bloques

Una expresión como `x = 0` ó `i++` o `printf(…)` se convierte en una *proposición* cuando va seguida de un punto y coma `;`, como en

```
x = 0;
i++;
printf(...);
```

En C, el punto y coma `;` es un terminador de proposición, en lugar de un separador, como lo es en un lenguaje tipo Pascal.

Las llaves `{` y `}` se emplean para agrupar declaraciones y proposiciones dentro de una *proposición compuesta* o *bloque*, de modo que son sintácticamente equivalentes a una proposición sencilla. Las llaves que encierran las proposiciones de una función son un ejemplo obvio; otros ejemplos son las llaves alrededor de proposiciones múltiples después de un `if`, `else`, `while` o `for`. (Pueden declararse variables dentro de cualquier bloque; esto se expondrá en el capítulo 4). No hay punto y coma después de la llave derecha que termina un bloque.

3.2 If-Else

La proposición `if-else` se utiliza para expresar decisiones. Formalmente, la sintaxis es

```
if (expresión)
    proposición1
else
   proposición2
```

donde la parte del `else` es optativa. La expresión se evalúa; si es verdadera (esto es, si la expresión tiene un valor diferente de cero), la *proposición1* se ejecuta. Si es falsa (*expresión* es cero) y si existe una parte de `else`, se ejecuta en su lugar la *proposición2*.

Puesto que un `if` simplemente prueba el valor numérico de una expresión, son posibles ciertas abreviaciones de código. Lo más obvio es escribir

```
if (expresión)
```

en lugar de

```
if (expresión != 0)
```

Algunas veces esto es claro y natural; otras puede ser misterioso.

Debido a que la parte `else` de un `if-else` es optativa, existe una ambigüedad cuando un `else` se omite de una secuencia `if` anidada. Esto se resuelve al asociar el `else` con el `if` sin `else` anterior más cercano. Por ejemplo, en

```
if (n > 0)
    if (a > b)
        z = a;
else
    z = b;
```

el `else` va con el `if` más interno, como se muestra con el sangrado. Si eso no es lo que se desea, se deben utilizar llaves para forzar la asociación correcta:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
4
```

La ambigüedad es especialmente perniciosa en situaciones como esta:

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
}
else    /* EQUIVOCADO */
    printf("error -- n es negativo\n");
```

El sangrado muestra en forma inequívoca lo que se desea, pero el compilador no entiende el mensaje y asocia el `else` con el `if` más interno. Puede ser difícil encontrar esta clase de errores; es una buena idea utilizar llaves cuando hay varios `if` anidados.

A propósito, nótese que hay un punto y coma `;` después de `z = a` en

```
if (a > b)
    z = a;
else
    z = b;
```

Esto se debe a que gramaticalmente al `if` le sigue una *proposición*, y una expresión como "`z = a;`" siempre se termina con punto y coma `;`.

3.3 Else-if

La construcción

```
if (expresión)
    proposición
else if (expresión)
    proposición
else if (expresión)
```

```
    proposición
else if (expresión)
    proposición
else
    proposición
```

ocurre de modo tan frecuente que bien vale una pequeña discusión aparte. Esta secuencia de proposiciones `if` es la forma más general de escribir una decisión múltiple. Las *expresiones* se evalúan en orden; si cualquier *expresión* es verdadera, se ejecuta la *proposición* asociada con ella, y esto termina toda la cadena. Como siempre, el código para cada *proposición* es una proposición simple o un grupo dentro de llaves.

La parte del último `else` maneja el caso “ninguno de los anteriores” o *caso por omisión* cuando ninguna de las otras condiciones se satisface. En algunos casos no hay una acción explícita para la omisión; en ese caso el último

```
else
    proposición
```

puede omitirse, o puede utilizarse para detección de errores al atrapar una condición “imposible” .

Para ilustrar una decisión de tres vías, se muestra a continuación una función de búsqueda binaria que decide si un valor particular de `x` se encuentra en el arreglo ordenado `v` - Los elementos de `v` deben estar en orden ascendente. La función regresa la posición (un número entre `0` y `n-1`) si `x` ocurre en `v`, y `-1` si no es así.

La búsqueda binaria primero compara el valor de entrada `x` con el elemento medio del arreglo `v`. Si `x` es menor que el valor del medio, la búsqueda se enfoca sobre la mitad inferior de la tabla; de otra manera lo hace en la mitad superior, cualquier caso, el siguiente paso es comparar a `x` con el elemento medio de la mitad seleccionada. Este proceso de dividir en dos continúa hasta que se encuentra el valor o ya no hay elementos.

```
/* binsearch: encuentra x en v[0] <= v[1] <= ... <= v[n-1] */
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high)/2;
        if (x < v[mid])
            high = mid + 1;
        else if (x > v[mid])
            low = mid + 1;
        else    /* el elemento fue encontrado */
            return mid;
    }
    return -1;    /* no fue encontrado */
}
```

La decisión fundamental es si `x` es menor que, mayor que o igual al elemento medio `v[mid]` en cada paso; esto es un `else-if` natural.

- **Ejercicio 3-1**. Nuestra búsqueda binaria realiza dos pruebas dentro del ciclo, cuando una podría ser suficiente (al precio de más pruebas en el exterior). Escriba una versión con sólo una prueba dentro del ciclo y mida la diferencia en tiempo de ejecución. □

3.4 Switch

La proposición `switch` es una decisión múltiple que prueba si una expresión coincide con uno de un número de valores *constantes* enteros, y traslada el control adecuadamente.

```
switch (expresión) {
    case exp-const: proposiciones
    case exp-const: proposiciones
    default: proposiciones
)
```

Cada `case` se etiqueta con uno o más valores constantes enteros o expresiones constantes enteras. Si un `case` coincide con el valor de la expresión, la ejecución comienza allí. Todas las expresiones `case` deben ser diferentes. El que está etiquetado como `default` se ejecuta sí ninguno de los otros se satisface. El `default` es optativo; si no está y ninguno de los casos coincide, no se toma acción alguna. Las cláusulas `case` y `default` pueden ocurrir en cualquier orden.

En el capítulo 1 se escribió un programa para contar las ocurrencias de cada dígito, espacio en blanco y todos los demás caracteres, usando una secuencia de `if … else if … else`. Aquí está el mismo programa con un `switch`:

```
#include <stdio.h>
main()    /* cuenta dígitos, espacios blancos, y otros */
{
        int c, i, nwhite, nother, ndigit[10];

        nwhite = nother = 0;
        for (i = 0; i < 10; i++)
            ndigit[i] = 0;
        while ((c = getchar()) != EOF) {
            switch (c) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                ndigit[c-'0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nwhite++;
                break;
            default:
                nother++;
                break;
            }
    }
```

```
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", espacio en blanco = %d, otros = %d\n",
        nwhite, nother);
    return 0;
}
```

La proposición `break` provoca una salida inmediata del `switch`. Puesto que los `case` sirven sólo como etiquetas, después de que se ejecuta el código para Uno, la ejecución *pasa al siguiente*, a menos que se tome una acción específica para terminar el `switch`. Las formas más comunes de dejar un `switch` son `break` y `return`. Una proposición `break` también se puede emplear para forzar una salida inmediata de los ciclos `while`, `for` y `do`, como se verá más adelante en este capítulo.

Pasar a través de los `case` es en parte bueno y en parte no. Por el lado positivo, esto permite conectar varios `case` a una acción simple, como con los dígitos de este ejemplo. Pero eso también implica que cada case normalmente debe terminar con un `break` para prevenir pasar al siguiente. Pasar de un `case` a otro no es una práctica muy sólida y es susceptible a la desintegración cuando se modifica el programa. Con la excepción de etiquetas múltiples para un cálculo simple, lo anterior se debe utilizar con cautela y emplear comentarios.

Como formalidad, coloque un `break` después del último `case` (en este caso el `default`) aun si desde ek punto de vista lógico resulta innecesario. Algún día - cuando se agregue otro `case` al final - esta práctica de programación defensiva lo salvará.

- **Ejercicio 3-2**. Escriba una función `escape(s,t)` que convierte caracteres como *nueva línea* y *tabulación* en secuencias de escape visibles como `\n` y `\t` mientras copia la cadena `t` a `s`. Utilice un `switch`. Escriba también una función para la dirección inversa, convirtiendo secuencias de escape en caracteres reales. □

3.5 Ciclos - While y For

Ya hemos encontrado los ciclos `while` y `for`. En

```
while (expresión)
    proposición
```

la expresión se evalúa. Si es diferente de cero, se ejecuta la *proposición* y se reevalúa la *expresión*. Este ciclo continúa hasta que la *expresión* se hace cero, punto en el cual se suspende la ejecución para continuar después de la *proposición*.

La proposición `for`

```
for (expr1; expr2; expr3)
    proposición
```

es equivalente a

```
expr1;
while (expr2) {
    proposición
    expr3;
}
```

excepto por el comportamiento de `continue` que se describe en la sección 3.7.

Gramaticalmente, las tres componentes de un ciclo `for` son expresiones. Por lo común, *expr1* y *expr3* son asignaciones o llamadas a función y *expr2* es una expresión de relación. Cualquiera de las tres partes se puede omitir, aunque deben permanecer los punto y coma `;`. Si se omite *expr1* o *expr3*, sólo se desecha de la expansión. Si la prueba *expr2* no está presente, se toma como permanentemente verdadera, así que

```
for (;;) {
    ...
}
```

es una iteración “infinita”, que presumiblemente será interrumpida por otros medios, como un `break` o un `return`.

El usar `while` o `for` es principalmente cuestión de preferencia personal. Por ejemplo, en

```
while ((c = getchar()) == ' ' || c == '\n' || c = '\t')
    ; /* ignora caracteres espaciadores */
```

no hay inicialización o reinicialización, por lo que el `while` es más natural.

El `for` se prefiere cuando existe una inicialización simple e incrementos, puesto que mantiene las proposiciones de control del ciclo juntas y visibles al principio del mismo. Esto es más obvio en

```
for (i = 0; i < n; i++)
    ...
```

que es la forma característica de procesar los primeros *n* elementos de un arreglo en C, lo análogo al ciclo `DO` de Fortran o al `for` de Pascal. Sin embargo, la analogía no es perfecta puesto que tanto el índice como el límite de un ciclo `for` en C pueden ser alterados desde dentro del ciclo, y la variable del Índice retiene su valor cuando por cualquier razón terminan las iteraciones. Puesto que las componentes del `for` son expresiones arbitrarias, sus ciclos no están restringidos a progresiones aritméticas. Por otra parte, considere que es un mal estilo incluir en las secciones de inicialización e incremento operaciones no relacionadas con esas actividades, que más bien se reservan para acciones de control del ciclo.

Como un ejemplo más amplio, aquí está otra versión de `atoi` para convertir una cadena a su equivalente numérico. Esta es ligeramente más general que la del capítulo 2; trata también los espacios en blanco previos al número, y los

signos + o -. (El capitulo 4 muestra `atof`, que realiza la misma conversión para números de punto flotante).

La estructura del programa refleja la forma de la entrada:

- ignora espacios en blanco, si los hay
- toma el signo, si lo hay
- toma la parte entera y conviértela

Cada paso realiza su parte, y deja las cosas en forma clara para el siguiente. La totalidad del proceso termina con el primer carácter que no pueda ser parte de un número.

```
#include <ctype.h>

/* atoi: convierte s a entero; versión 2 */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)    /* ignora espacio en blanco * /
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')    /* ignora el signo */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

La biblioteca estándar proporciona una función más elaborada, `strtol`, para la conversión de cadenas a enteros largos; véase la sección 5 del apéndice B.

Las ventajas de mantener centralizado el control del ciclo son aún más obvias cuando existen ciclos anidados. La siguiente función es una clasificación Shell para ordenar un arreglo de enteros. La idea básica de este algoritmo de ordenamiento - inventado en 1959 por D.L. Shell - es que en las primeras etapas sean comparados elementos lejanos (en lugar de los adyacentes, como en los ordenamientos de intercambio más simples). Esto tiende a eliminar rápidamente gran cantidad de desorden, así que los estados posteriores tienen menos trabajo por hacer. El intervalo entre los elementos comparados disminuye en forma gradual hasta uno, punto en el que el ordenamiento viene a ser efectivamente un método adyacente de intercambio.

```
/* shellsort: ordena v[0]...v[n-1] en orden ascendente */
void shellsort(int v[], int n)
{
    int gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
        for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
            temp = v[j];
            v[j] = v[j+gap];
            v[j+gap] = temp;
        }
```

```
}
```

Existen tres ciclos anidados. El más externo controla el espacio entre los elementos comparados, reduciéndolo desde `n/2` por un factor de dos en cada paso hasta que llega a cero. El ciclo intermedio recorre los elementos. El ciclo más interno compara cada pareja de elementos que está separada por el espacio `gap` e invierte a las que estén desordenadas. Puesto que `gap` finalmente se reduce a uno, todos los elementos se ordenan correctamente. Nótese cómo la generalidad del `for` hace que el ciclo más externo coincida con la forma de los otros, aun cuando no es una progresión aritmética.

Un último operador de C es la coma `,` que frecuentemente encuentra uso en la proposición `for`. Una pareja de expresiones separadas por una coma se evalúa de izquierda a derecha, y el tipo y valor del resultado son el tipo y valor del operando derecho. Así, en una proposición `for` es posible colocar expresiones múltiples en las diferentes partes, por ejemplo, para procesar dos índices en paralelo. Esto se ilustra en la función `reverse(s)`, que invierte a la cadena `s` en el mismo lugar.

```
#include <string.h>

/* reverse: invierte la cadena s en el mismo lugar */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Las comas que separan a los argumentos de una función, las variables en declaraciones, etc., *no son* operadores coma, y no garantizan evaluación de izquierda a derecha.

Los operadores coma deberán utilizarse poco. Los usos más adecuados son en construcciones fuertemente relacionadas una con la otra, como en el ciclo `for` de `reverse`, y en macros en donde un cálculo de paso múltiple debe ser una expresión simple. Una expresión coma podría también ser apropiada para el intercambio de elementos en `reverse`, donde el intercambio puede ser a través de una operación simple:

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

- **Ejercicio 3-3**. Escriba la función `expand(sl,s2)` que expande notación abreviada como `a-z`, que viene en la cadena `s1`, en la lista equivalente completa `abc…xyz` en `s2`. Permita letras mayúsculas y minúsculas, así como dígitos, y esté preparado para manejar casos como `a-b-c` y `a-z0-9`

y `–a–z`. Haga que los guiones `-` al inicio o al final se tomen literalmente. □

3.6 Ciclos - do-while

Como ya se expuso en el capítulo 1, los ciclos `while` y `for` verifican al principio la condición de término. En contraste, el tercer ciclo en C, el `do-while`, prueba al final *después* de realizar cada paso a través del cuerpo del ciclo, el cual se ejecuta siempre por lo menos una vez.

La sintaxis del do es

```
do
    proposición
while (expresión);
```

La *proposición* se ejecuta y después se evalúa la *expresión*. Si es verdadera, la *proposición* se evalúa de nuevo, y así sucesivamente. Cuando la expresión se hace falsa, el ciclo termina. Excepto por el sentido de la prueba, el `do-while` es equivalente a la proposición `repeat-until` de Pascal.

La experiencia demuestra que el `do-while` es mucho menos utilizado que el `while` y el `for`. Aunque de cuando en cuando es valioso, como en la siguiente función `itoa`, que convierte un número a una cadena de caracteres (lo inverso de `atoi`). El trabajo es ligeramente más complicado de lo que podría pensarse en un principio, debido a que los métodos fáciles para generar dígitos los generan en el orden incorrecto. Hemos elegido generar la cadena al revés y después invertirla.

```
/* itoa: convierte n a caracteres en s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0) /* registra el signo */
        n = -n;         /* hace a n positivo */
    i = 0;
    do {      /* genera digitos en orden inverso */
        s[i++] = n % 10 + '0'; /* obtiene siguiente dígito */
    } while ((n /= 10) > 0);   /* lo borra */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
reverse(s);
}
```

El `do-while` es necesario, o al menos conveniente, puesto que por lo menos se debe instalar un carácter en el arreglo `s`, aun si `n` es cero. También empleamos llaves alrededor de la proposición simple que hace el cuerpo del do-while, aunque son innecesarias, y así el lector apresurado no confundirá la sección del `while` con el *comienzo* de un ciclo `while`.

- **Ejercicio 3-4**. En una representación de números en complemento a dos, nuestra versión de `itoa` no maneja el número negativo más grande,

esto es, el valor de `n` igual a `-( 2^tamañopalabra-1)`. Explique por qué. Modifíquelo para imprimir el valor correctamente, sin importar la máquina en que ejecute. □

- **Ejercicio 3-5**. Escriba la función `itob(n,s,b)` que convierte al entero `n` en una representación de caracteres con base `b` dentro de la cadena `s`. En particular, `itob(n,S,16)` da formato a `s` como un entero hexadecimal en `s`. □
- **Ejercicio 3-6**. Escriba una versión de `itoa` que acepte tres argumentos en lugar de dos. El tercer argumento es un ancho mínimo de campo; si es necesario, al número convertido debe agregársele caracteres en blanco a la izquierda para hacerlo lo suficientemente ancho. □

3.7 Break y Continue

Algunas veces es conveniente tener la posibilidad de abandonar un ciclo de otra manera que no sea probando al inicio o al final. La proposición `break` proporciona una salida anticipada de un `for`, `while` y `do`, tal como lo hace el `switch`.

Un `break` provoca que el ciclo o `switch` más interno que lo encierra termine inmediatamente.

La siguiente función, `trim`, elimina espacios blancos, tabuladores y nuevas líneas al final de una cadena, utilizando un `break` para salir de un ciclo cuando se encuentra el no-blanco, no-tabulador o no-nueva línea de más a la derecha.

```
/* trim: elimina blancos, tabuladores y nueva linea al final */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

`strlen` regresa la longitud de la cadena. El ciclo `for` inicia al final y rastrea hacia atrás, buscando el primer carácter que no sea blanco o tabulador o nueva linea.

El ciclo se interrumpe cuando se encuentra alguno o cuando *n* se hace negativa (esto es, cuando se ha rastreado toda la cadena. Se deberá verificar que este comportamiento es correcto, aun cuando la cadena esté vacía o sólo contiene espacios en blanco.

La proposición `continue` está relacionada con el `break`, pero se utiliza menos; provoca que inicie la siguiente iteración del ciclo `for`, `while` o `do` que la contiene. Dentro de `while` y `do`, esto significa que la parte de la prueba se ejecuta inmediatamente; en el `for`, el control se traslada al paso de incremento. La proposición continué se aplica solamente a ciclos, no a `switch`.

Un `continue` dentro de un `switch` que está a su vez en un ciclo, provoca la siguiente iteración del ciclo.

Como un ejemplo, el siguiente fragmento procesa sólo los elementos no negativos que están en el arreglo a; los valores negativos son ignorados.

```
for (i = 0; i < n; i++)
    if (a[i] < 0)    /* ignora elementos negativos */
        continue;
... /* trabaja elementos positivos */
```

La proposición `continue` se emplea a menudo cuando la parte del ciclo que sigue es complicada, de modo que invertir la prueba y sangrar otro nivel podría anidar profundamente el programa.

3.8 Goto y Etiquetas

C proporciona la infinitamente abusable proposición `goto`, y etiquetas para saltar hacia ellas. Formalmente, el `goto` nunca es necesario, y en la práctica es casi siempre más fácil escribir código sin él. En este libro no se ha usado `goto` alguno.

Sin embargo, hay algunas situaciones donde los `goto` pueden encontrar un lugar. La más común es abandonar el procesamiento en alguna estructura profundamente anidada, tal como salir de dos o más ciclos a la vez. La proposición `break` no se puede utilizar directamente, puesto que sólo sale del ciclo más interno. Así:

```
    for ( ... )
        for ( ... ) {
            ...
            if (desastre)
                goto error;
        }
    ...
error:
    /* arregla el desorden /*
```

Esta organización es útil si el código de manejo de error no es trivial y si los errores pueden ocurrir en varios lugares.

Una etiqueta tiene la misma forma que un nombre de variable y es seguida por dos puntos `:`. Puede ser adherida a cualquier proposición de la misma función en la que está el `goto`. El alcance de una etiqueta es toda la función.

Como otro ejemplo, considérese el problema de determinar si dos arreglos, `a` y `b`, tienen un elemento en común. Una posibilidad es

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto encontrado;
    /* no se encontró ningún elemento común */
    ...
encontrado:
```

```
    /* se encontró uno: a[i] == b[j] */
    ...
```

El código que involucra un `goto` siempre puede escribirse sin él, aunque tal vez al precio de algunas pruebas repetidas o variables extra. Por ejem plo, la búsqueda en los arreglos quedará

```
encontrado = 0;
for (i = 0; i < n && !encontrado; i++)
    for (j = 0; j < m && !encontrado; j++)
        if (a[i] == b[j])
            encontrado = 1;
if (encontrado)
    /* Se encontró uno: a[i-1] == b[j-1] */
    ...
else
    /* no se encontró ningún elemento común */
    ...
```

Con pocas excepciones, como las citadas aquí, el código que se basa en proposiciones `goto` es generalmente más difícil de entender y de mantener que el código sin ellas. Aunque no somos dogmáticos acerca del asunto, se ve que las proposiciones `goto` deben ser utilizadas raramente, si acaso.

Capitulo 4: Funciones y la estructura del programa

Las funciones dividen tareas grandes de computación en varias más pequeñas, y permiten la posibilidad de construir sobre lo que otros ya han hecho, en lugar de comenzar desde cero. Las funciones apropiadas ocultan los detalles de operación de las partes del programa que no necesitan saber acerca de ellos, así que dan claridad a la totalidad y facilitan la penosa tarea de hacer cambios.

El lenguaje C se diseñó para hacer que las funciones fueran eficientes y fáciles de usar; los programas escritos en C se componen de muchas funciones pequeñas en lugar de sólo algunas grandes. Un programa puede residir en uno o más archivos fuente, los cuales pueden compilarse por separado y cargarse junto con funciones de biblioteca previamente compiladas. No trataremos aquí tales procesos, puesto que los detalles varían de un sistema a otro.

La declaración y definición de funciones es el área donde el estándar ANSI ha hecho los cambios más visibles a C. Tal como mencionam os en el capítulo 1, ahora es posible declarar los tipos de los argumentos cuando se declara una función. La sintaxis de la definición de funciones también cambia, de modo que las declaraciones y las definiciones coincidan. Esto hace posible que el compilador pueda detectar muchos más errores de lo que podía anteriormente. Además, cuando los argumentos se declaran con propiedad, se realizan automáticamente las conversiones convenientes.

El estándar clarifica las reglas sobre el alcance de los nombres; en particular, requiere que sólo haya una definición de cada objeto externo. La inicialización es más general: los arreglos y las estructuras automáticas ahora se pueden inicializar.

El preprocesador de C también se ha mejorado. Las nuevas facilidades del Procesador incluyen un conjunto más completo de directivas para la compilación condicional, una forma de crear cadenas entrecomilladas a partir de argumentos de macros y un mejor control sobre el proceso de expansión de macros.

4.1 Conceptos básicos de funciones

Para comenzar, diseñemos y escribamos un programa que imprima cada línea de su entrada que contenga un “patrón” o cadena de caracteres en particular.

(Este es un caso especial del programa *grep* de UNIX.) Por ejemplo, al buscar el patrón de letras “ould.” en el conjunto de líneas

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
```

```
Re-mould it nearer to the Heart's Desire!
```

producirá la salida

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

El trabajo se ajusta ordenadamente en tres partes:

```
while {hay otra línea)
   if {la línea contiene el patrón)
      imprímela
```

Aunque ciertamente es posible poner el código de todo esto en `main`, una mejor forma es aprovechar la estructura haciendo de cada parte una función separada. Es más fácil trabajar con tres piezas pequeñas que con una grande, debido a que los detalles irrelevantes se pueden ocultar dentro de las funciones, y minimizar así el riesgo de interacciones no deseadas. Los fragmentos incluso se pueden emplear en otros programas.

“Mientras hay otra línea” es `getline`, función que ya escribimos en el capítulo 1, e “imprímela” es `printf`, que alguien ya nos proporcionó. Esto significa que sólo necesitamos escribir una rutina para decidir si la línea contiene una ocurrencia del patrón.

Podemos resolver ese problema escribiendo una función `strindex(s,t)`, que regresa la posición o índice en la cadena `s` en donde comienza la cadena `t`, o `-1` si `s` no contiene `t`. Debido a que los arreglos en C principian en la posición cero, los índices serán cero o positivos, y así un valor negativo como `-1` es conveniente para señalar una falla. Cuando posteriormente se necesite una coincidencia de patrones más elaborada, sólo se debe reemplazar `strindex`; el resto del código puede permanecer igual. (La biblioteca estándar provee una función `strstr` que es semejante a `strindex`, excepto en que regresa un apuntador en lugar de un índice.)

Una vez definido todo este diseño, llenar los detalles del programa es simple. Aquí está en su totalidad, de modo que se puede ver cómo las piezas quedan juntas. Por ahora, el patrón que se buscará es una cadena literal, lo cual no es el mecanismo más general. Regresaremos en breve a una discusión sobre cómo inicializar arreglos de caracteres, y en el capítulo 5 mostraremos cómo hacer que el patrón de caracteres sea un parámetro fijado cuando se ejecuta el programa.

También hay una versión ligeramente diferente de `getline`, que se podrá comparar con la del capítulo 1.

```
# include <stdio.h>
#define MAXILINE 1000    /* longitud máxima por línea de entrada */

int getline(char line[], int max)
int strindex(char source[], char searchfor[]);
```

```
char pattern[] = "ould";     /* patrón a buscar */

/* encontrar todas las líneas que coincidan con el patrón */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: pone linea en s, retorna longitud */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: retorna el índice de t en s, -1 si no hay */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Cada definición de función tiene la forma

```
tipo-regresado nombre-de-función(declaraciones de argumentos)
{
    declaraciones y proposiciones
}
```

Varias partes pueden omitirse; una función mínima “dummy” es

```
nada() {}
```

que no hace ni regresa nada. Una función `hacer-nada`, como ésta, es algunas veces útil para reservar lugar al desarrollar un programa. Si el tipo que regresa se omite, se supone `int`.

Un programa es sólo un conjunto de definiciones de variables y funciones. La comunicación entre funciones es por argumentos y valores regresados por las funciones, y a través de variables externas. Las funciones pueden presentarse en cualquier orden dentro del archivo fuente, y el programa fuente se puede dividir en varios archivos, mientras las funciones no se dividan.

La proposición `return` es el mecanismo para que la función que se llama regrese un valor a su invocador. Al `return` le puede seguir cualquier expresión:

```
return expresión
```

La *expresión* se convertirá al tipo de retorno de la función si es necesario. Con frecuencia se utilizan paréntesis para encerrar la (*expresión*), pero son optativos.

La función que llama tiene la libertad de ignorar el valor regresado. Incluso, no hay necesidad de un a expresión después de `return`; en tal caso, ningún valor regresa al invocador. También el control regresa, sin valor, cuando la ejecución "cae al final" de la función al alcanzar la llave cerrada derecha `}`. No es ilegal, aunque probablemente un signo de problemas, el que una función regrese un valor desde un lugar y ninguno desde otro. En cualquier caso, si una función no regresa explícitamente un valor, su "valor" es ciertamente basura.

El programa de búsqueda del patrón regresa un estado desde `main`, el número de coincidencias encontradas. Este valor está disponible para ser empleado por el medio ambiente que llamó al programa.

El mecanismo de cómo compilar y cargar un programa en C que reside en varios archivos fuente varía de un sistema a otro. En el sistema UNIX, por ejemplo, la orden *cc* mencionada en el capítulo 1 hace el trabajo. Suponiendo que las tres funciones se almacenan en tres archivos llamados `main.c`, `getline.c`, y `strindex.c`. Entonces la orden

```
cc main.c getline.c strindex.c
```

compila los tres archivos, sitúa el código objeto resultante en los archivos `main.o`, `getline.o`, y `strindex.o`, y después los carga todos dentro de un archivo ejecutable llamado `a.out`. Si existe un error - digamos en `main.c` - dicho archivo puede volverse a compilar por sí mismo y el resultado cargado con los archivos objeto previos, con la orden.

```
cc main.c getline.o strindex.o
```

cc emplea la convención "`.c`" contra "`.o`" para distinguir los archivos fuente de los archivos objeto.

- **Ejercicio 4-1**. Escriba la función `strrindex(s,t)`, que regresa la posición de la ocurrencia de más a la derecha de `t` en `s`, o `-1` si no hay alguna.

4.2 Funciones que regresan valores no enteros

Basta ahora los ejemplos de funciones han regresado o ningún valor (void) o un `int`. ¿Qué pasa si una función debe regresar algo de otro tipo? Muchas funciones numéricas como `sqrt`, `sin` y `cos` regresan `double`; otras funciones especializadas regresan tipos diferentes. Para ilustrar cómo tratar con esto, escribamos y usemos la función `atof(s)`, que convierte la cadena `s` a su valor equivalente de punto flotante de doble precisión. La función `atoi` es una extensión de `atoi`, de la que mostramos versiones en los capítulos 2 y 3. Maneja signo y punto decimal optativos, y presencia o ausencia de parte entera o fraccionaria. Nuestra versión no es una rutina de conversión de alta calidad; tomaría más espacio del que podemos dedicarle. La biblioteca estándar incluye un `atof`; el *header* `<math.h>` la declara.

Primero, `atof` por sí misma debe declarar el tipo del valor que regresa, puesto que no es `int`. El nombre del tipo precede al nombre de la función:

```
#include <ctype.h>

/* atof: convierte la cadena s a double */
double atof(char s[])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++) /* saltea espacio en blanco */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    return sign * val / power;
}
```

Segundo, e igualmente importante, la rutina que llama debe indicar que `atof` regresa un valor que no es `int`. Una forma de asegurar esto es declarar `atof` explícitamente en la rutina que la llama. La declaración se muestra en esta primitiva calculadora (apenas adecuada para un balance de chequera), que lee un número por línea, precedido en forma optativa por un signo, y lo acumula, imprimiendo la suma actual después de cada entrada:

```
#include <stdio.h>
#define MAXLINE 100    /* calculadora rudimentaria * /
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);
```

```
    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

La declaración

```
double sum, atof(char []);
```

señala que `sum` es una variable `double`, y que `atof` es una función que toma un argumento `char[]` y regresa un `double`.

La función `atof` se debe declarar y definir consistentemente. Si `atof` en sí misma y la llamada a ella en `main` tienen tipos inconsistentes dentro del mismo archivo fuente, el error será detectado por el compilador. Pero si (como es probable) `atof` fuera compilada separadamente, la falta de consistencia no se detectaría, `atof` regresaría un valor `double` que `main` trataría como `int`, y se producirían resultados incongruentes.

A la luz de lo que hemos mencionado acerca de cómo deben coincidir las declaraciones con las definiciones, esto podría ser sorprendente. La razón de que ocurra una falta de coincidencia es que, si no existe el prototipo de una función, ésta es declarada implícitam ente la primera vez que aparece en una expresión, como

```
sum += atof(line)
```

Si en una expresión se encuentra un nombre que no ha sido declarado previamente y está seguido por paréntesis izquierdo, se declara por contexto, de modo que se supone que es el nombre de una función que regresa un `int`, y nada se supone acerca de sus argumentos. Aún más, si la declaración de una función no incluye argumentos como en

```
double atof();
```

también es tomada de modo que no se supone nada acerca de los argumentos de `atof`; se desactiva toda revisión de parámetros. Este significado especial de la lista de argumentos vacía se hace para permitir que los programas en C viejos se compilen con los nuevos compiladores. Pero es una mala táctica usar esto con programas nuevos. Si la función toma argumentos, declárelos; si no los toma, use `void`.

Dado `atof`, propiamente declarado, podemos escribir `atoi` (convierte una cadena a `int`) en términos de él:

```
/* atoi: convierte la cadena s a entero usando atof */
int atoi(char s[])
{
double atof(char s[]);
return (int) atof(s);
}
```

Nótese la estructura de las declaraciones y la proposición `return`. El valor de la expresión en

```
return expresión;
```

se convierte al tipo de la función antes de que se tome el `return`. Por lo tanto, el valor de `atof`, un `double`, se convierte automáticamente a `int` cuando aparece en este `return`, puesto que la función `atoi` regresa un `int`. Sin embargo, esta operación potencialmente descarta información, de manera que algunos compiladores lo previenen. El *cast* establece explícitamente lo que la operación intenta y suprime las advertencias.

- **Ejercicio 4-2**. Extienda `atof` para que maneje notación científica de la forma `123.45e-6` donde un número de punto flotante puede ir seguido por `e` o `E` y opcionalmente un exponente con signo. □

4.3 Variables Externas

Un programa en C consta de un conjunto de objetos externos, que son variables o funciones. El adjetivo “externo” se emplea en contraste con “interno”, el cual describe los argumentos y las variables definidas dentro de las funciones. Las variables externas se definen fuera de cualquier función, y por lo tanto , están potencialmente disponibles para muchas funciones. Las funciones en sí mismas son siempre externas, puesto que C no permite definir funciones dentro de otras funciones. Por omisión, las variables y funciones externas tienen la propiedad de que todas las referencias a ellas por el mismo nombre, incluso desde funciones compiladas separadamente, son referencias a la misma cosa. (El estándar llama a esta propiedad *ligado externo*). En este sentido, las variables externas son análogas a los bloques `COMMON` de Fortran o a las variables del bloque más externo de Pascal. Más adelante veremos cómo definir variables y funciones externas que sean visibles sólo dentro de un archivo fuente.

Debido a que las variables externas son accesibles globalmente, proporcionan una alternativa a los argumentos en funciones y a los valores de retorno para comunicar datos entre funciones. Cualquier función puede tener acceso a variables externas haciendo referencia a ellas solamente por su nombre, si éste ha sido declarado de alguna manera.

Si un gran número de variables se debe compartir entre funciones, las variables externas son más convenientes y eficientes que las largas listas de argumentos. Sin embargo, como se señaló en el capítulo 1, este razonamiento se deberá aplicar con precaución, pues puede tener un efecto negativo sobre la estructura del programa y dar lugar a programas con demasiadas conexiones de datos entre funciones.

Las variables externas son también útiles debido a su mayor alcance y tiempo de vida. Las variables automáticas son internas a una función y su existencia se inicia cuando se entra a la función y desaparecen cuando ésta se abandona.

Por otro lado, las variables externas son permanentes, de modo que retienen sus valores de la invocación de una función a la siguiente. Así, si dos funciones deben compartir algunos datos, aun si ninguna llama a la otra, con frecuencia es más conveniente que los datos compartidos se mantengan en variables externas, en lugar de que sean pasados como argumentos de entrada y salida.

Examinemos más a fondo este tema con un ejemplo más amplio. El problema es escribir el programa de una calculadora que provea los operadores `+ - * y /`. Por ser más fácil su implantación, la calculadora utilizará notación polaca inversa en lugar de infija. (La polaca inversa es utilizada por algunas calculadoras de bolsillo, y en lenguajes como Forth y PostScript.)

En notación polaca inversa, cada operador sigue a sus operandos; por lo tanto, lo que en una expresión infija se expresa como

```
(1 - 2) * (4 + 5)
```

en notación polaca inversa se introduce como

```
1 2 - 4 5 + *
```

Las paréntesis son innecesarias; toda vez que sepamos cuántos operandos espera cada operador, la notación no resulta ambigua.

La implantación es simple. Cada operando se introduce en una pila o “stack”, cuando se arriba a un operador, se extrae el número correcto de operandos (dos en el caso de los operadores binarios), se aplica el operador y el resultado se regresa a la pila. En el ejemplo anterior, se introducen `1` y `2`, luego se reemplazan por su diferencia, `-1`. A continuación se introducen `4` y `5` y luego se reemplazan por su adición, `9`. El producto de `-1` y `9`, que es `-9`, los reemplaza en la pila. El valor que se encuentra en el tope de la pila se extrae e imprime cuando se encuentra el fin de la línea de entrada.

De esta forma, La estructura del programa que opere de esta manera será un ciclo que realice las operaciones adecuadas sobre cada operador y operando en la medida que aparecen;

```
while (siguiente operador u operando que no es fin de archivó)
    if (número)
        introducirlo
    else if (operador)
        extraer operandos
        hacer operaciones
        introducir el resultado
    else if (nueva línea)
        extrae e imprime el tope de la pila
    else
        error
```

Las operaciones de introducir en pila (*push*, o “apilar”) y extraer de una pila (*pop* o “desapilar”) resultan triviales, pero cuando al intentar agregar la detección y recuperación de errores, resultan suficientemente largas como para que sea conveniente colocarlas en funciones separadas en lugar del

código repartido a lo largo de todo el programa. Además, debería existir una función separada para buscar el siguiente operador u operando.

La principal decisión de diseño que aún no se ha explicado es dónde está la pila, esto es, cuáles rutinas tienen acceso a ella directamente. Una posibilidad es mantenerla en `main`, y pasar la pila y la posición actual a las rutinas que introducen y extraen elementos. Pero `main` no necesita saber acerca de las variables que controlan a la pila; sólo efectúa las operaciones de introducir y extraer. Así, hemos decidido almacenar la pila y su información asociada en variables externas accesibles a las funciones `push` y `pop`, pero no a main.

Traducir este bosquejo a código es sumamente fácil. Si por ahora pensamos que el programa existe en un archivo fuente, se verá así

```
#includeS
#defineS

declaración de funciones para main

main() { ... }

variables externas para push y pop

  void push(double f) { ... }
  double pop(void) { ... }

  int getop(char s[ ]) { ... }
rutinas llamadas por getop
```

Más adelante se verá cómo se puede dividir todo esto entre dos o más archivos de código fuente.

La función `main` es un ciclo que contiene un `switch` gigante sobre el tipo de operador y operando; éste es un uso del `switch` más típico que el mostrado en la sección 3.4.

```
#include <stdio.h>
#include <math.h>    /*para atof() */

#define MAXOP 100    /* máx tamaño de operando u operador */
#define NUMBER '0'   /* señal de que un número se encontró */

int getop(char []);
void push(double);
double pop(void);

/* Calculadora polaca inversa */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
```

```
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
            push(pop() / op2);
        else
            printf("error: divisor cero\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: comando desconocido %s\n", s);
            break;
        }
    }
    return 0;
}
```

Puesto que + y * son operadores conmutativos, el orden en el que se combinan los operandos extraídos es irrelevante (“el orden de los factores no altera el producto”), pero en el caso de `-` y `/`, se hace necesario distinguir su órden, en este caso específico, cuales son los operandos izquierdo y derecho. En

```
push(pop() - pop());   /* INCORRECTO */
```

no se ha definido el orden en el que se evalúan las dos llamadas de `pop`. Para garantizar que el orden sea correcto, se hace necesario extraer el primer valor en una variable temporal, como hicimos en `main`.

```
#define MAXVAL 100    /* máxima profundidad de la pila val •/
int sp = 0 ;          /* siguiente posición libre en la pila */
double val[MAXVAL];   /* valores de la pila */

/* push: introduce f a la pila */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: pula llena, no puede hacer push de %g\n", f);
}

/* pop: extrae y regresa el valor superior de la pila */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: pila vacia\n");
        return 0.0;
```

```
    }
}
```

Una variable es externa si se encuentra definida fuera de cualquier función. Por tanto, la pila y el índice de la pila que deben ser compartidos por `push` y por `pop` va definida por fuera de estas funciones. Pero `main` en sí misma no hace referencia a la pila o a la posición de la pila — la representación puede estar oculta.

Pasemos ahora a la implantación de `getop`, la función que toma el siguiente operador u operando. La tarea es fácil: “Ignorar blancos y tabuladores. Si el siguiente carácter no es un dígito o punto decimal, regresarlo; De otra manera, reunir una cadena de dígitos (que pueda incluir un punto decimal), y regresar `NUMBER` (la señal de que ha sido reunido un número)”.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: obtiene siguiente caracter u operando numérico */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;     /* no-número */
    i = 0;
    if (isdigit(c))   /* recoge parte de entero */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')     /* recoge parte fraccional */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

¿Qué son `getch` y `ungetch`? A menudo se da el caso de que un programa no puede determinar si ha leído lo suficiente de la entrada hasta que ya ha leído demasiado de ella. Reunir los caracteres que forman un número es un ejemplo de ello: el número no está completo hasta que se ve el primer caracter no-dígito… pero para cuando ello sucede, el programa ya ha leído un carácter de más, para lo cual no está preparado.

El problema podría ser resuelto si fuera posible “desleer” el carácter indeseado. De esta manera, toda vez que el programa lea un carácter de más, podría devolverlo a la entrada, provisión con la cual el resto del código se comportará como la lectura de más jamás hubiese sucedido. Afortunadamente, es fácil simular el regreso de un carácter, escribiendo un

par de funciones cooperativas, `getch` entrega el siguiente carácter de la entrada que va a ser considerado; `ungetch` reintegra el carácter devuelto a la entrada, de modo que llamadas posteriores a `getch` lo regresarán antes de leer algo nuevo de la entrada.

Cómo trabajan juntas es sencillo, `ungetch` coloca el carácter devuelto en un buffer compartido: un arreglo de caracteres. `getch` lee del buffer si hay algo allí, y si el buffer está vacío llama a `getchar`. También debe existir una variable índice que registra la posición del carácter actual en el buffer temporal. Puesto que el buffer y el índice son compartidos por `getch` y `ungetch` - y deben retener sus valores entre llamadas - deben ser externos a ambas rutinas. Asi, podemos escribir `getch`, `ungetch` y sus variables compartidas como:

```
#define BUFSIZE 100

char buf[BUFSIZE];  /* buffer para ungetch */
int bufp = 0;       /* siguiente posición libre en el buffer */

int getch(void)     /* obtiene un (posiblemente regresado) caracter */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)    /* devuelve caracter a la entrada */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: demasiados caracteres\n");
    else
        buf[bufp++] = c;
}
```

La biblioteca estándar incluye una función `ungetc` que proporciona el regreso de un carácter; esto se verá en el capitulo 7. Se ha utilizado un arreglo para lo que se regresa a la entrada, en lugar de un carácter sencillo, para dar una idea más general.

- **Ejercicio 4-3**. Dada la estructura básica, es fácil extender la calculadora. Agregue el operador módulo (`%`) y consideraciones para números negativos. □
- **Ejercicio 4-4**. Agregue órdenes para imprimir el elemento al tope de la pila sin sacarlo de ella, para duplicarlo y para intercambiar los dos elementos del tope. Agregue una orden para limpiar la pila. □
- **Ejercicio 4-5**. Agregue acceso a funciones de biblioteca como `sin`, `exp` y `pow`.

Consulte `<math.h>` en el apéndice B, sección 4. □

- **Ejercicio 4-6**. Agregue órdenes para manipular variables. (Es fácil proporcionar veintiséis variables con nombres de una letra.) Añada una variable para el valor impreso más reciente. □

- **Ejercicio 4-7**. Escriba un a rutina `ungets(s)` que regresa a la entrada una cadena completa. ¿Debe `ungets` conocer acerca de `buf` y `bufp`, o sólo debe usar `ungetch`? □
- **Ejercicio 4-8**. Suponga que nunca existirá más de un carácter de regreso. Modifique `getch` y `ungetch` de acuerdo con eso. □
- **Ejercicio 4-9**, Nuestros `getch` y `ungetch` no manejan correctamente un que se regresa. Decida cuáles deben ser sus propiedades si se regresa un , y después realice su diseño. □
- **Ejercicio 4-10**. Una organización alternativa emplea `getline` para leer una línea completa de entrada; esto hace innecesarios a `getch` y a `ungetch`. Corrija la calculadora para que use este planteamiento. □

4.4 Reglas y Alcance

4.5 Archivo de encabezamiento header

4.6 Variables estáticas

4.7 Variables tipo registro

4.8 Estructura de bloques

4.9 Inicialización

4.10 Recursividad

4.11 El preprocesador de C

4.1.1 Inclusión de archivos

4.1.2 Substitución de macros

4.1.3 Inclusión condicional